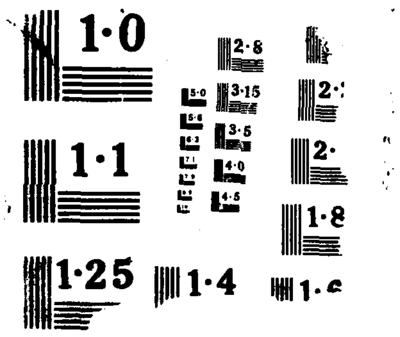
AD-A193 469 FORMAL HIERARCHICAL MULTILEVEL VERIFICATION OF SYNCHRONOUS MOS VLSI DESIGNS(U) MASSACHUSETTS INST OF TECH CAMBRIDGE MICROSYSTEMS RESEARCH CENTER D W WEISE 1/2

UNCLASSIFIED NOV 87 VLSI-MEMO-87-425 F/G 9/1 NL


1·0
1·1
1·25
1·4
1·6
1·8
2·8
3·15
3·5
4·0
4·5
5·0

MASSACHUSETTS INSTITUTE OF TECHNOLOGY • VLSI PUBLICATIONS

VLSI Memo No. 87-425
November 1987

FORMAL HIERARCHICAL MULTILEVEL VERIFICATION OF SYNCHRONOUS MOS VLSI DESIGNS

Daniel Wayne Weise

Abstract

I have designed and implemented a system for the multilevel verification of synchronous MOS VLSI circuits. The system, called Silica Pithecus, accepts the schematic of an MOS circuit and a specification of the circuit's intended digital behavior. Silica Pithecus determines if the circuit meets its specification. If the circuit fails to meet its specification Silica Pithecus returns to the designer the reason for the failure. Unlike earlier verifiers which modelled primitives (e.g., transistors) as unidirectional digital devices, Silica Pithecus models primitives more realistically. Transistors are modelled as bidirectional devices of varying resistances, and nodes are modelled as capacitors. Silica Pithecus operates hierarchically, interactively, and incrementally.

Major contributions of this research include a formal understanding of the relationship between different behavioral descriptions (e.g., signal, boolean, and arithmetic descriptions) of the same device, and a formalization of the relationship between the structure, behavior, and context of a device. Given these formal structures, my methods find sufficient conditions on the inputs of circuits which guarantee the correct operation of the circuit in the desired descriptive domain. These methods are algorithmic and complete. They also handle complex phenomena such as races and charge sharing. Informal notions such as races and hazards are shown to be derivable from the correctness conditions used by my methods.

Microsystems Research Center Room 39-321
Massachusetts Institute of Technology
Cambridge Massachusetts 02139
Telephone (617) 253-8138

Acknowledgements

Submitted to the Department of Electrical Engineering and Computer Science, August 1986, in partial fulfillment of the requirements for the degree of Doctor of Philosophy. This work was supported in part by the Defense Advanced Research Projects Agency under contract numbers N00014-86-K-0180, N00014-85-K-0124 and N00014-80-C-0622.

Author Information

Weise, current address: Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

Formal Multilevel Hierarchical Verification Of Synchronous MOS VLSI Circuits

by

Daniel Wayne Weise

B.S., University of California at Los Angeles
(1979)

S.M., Massachusetts Institute of Technology
(1982)

Submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

at the
Massachusetts Institute of Technology
August, 1986

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
August 15, 1986

Certified by Gerald Jay Sussman
Gerald J. Sussman
Professor of Electrical Engineering and Computer Science

Accepted by
Professor Arthur Smith
Chairman, Department Committee on Graduate Students

Formal Multilevel Hierarchical Verification Of Synchronous MOS VLSI Circuits

by

Daniel Wayne Weise

Submitted to the Department of Electrical Engineering
and Computer Science on August 15, 1986,
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy in Computer Science

Abstract

I have designed and implemented a system for the multilevel verification of synchronous MOS VLSI circuits. The system, called Silica Pithecus, accepts the schematic of an MOS circuit and a specification of the circuit's intended digital behavior. Silica Pithecus determines if the circuit meets its specification. If the circuit fails to meet its specification Silica Pithecus returns to the designer the reason for the failure. Unlike earlier verifiers which modelled primitives (e.g., transistors) as unidirectional digital devices, Silica Pithecus models primitives more realistically. Transistors are modelled as bidirectional devices of varying resistances, and nodes are modelled as capacitors. Silica Pithecus operates hierarchically, interactively, and incrementally.

Major contributions of this research include a formal understanding of the relationship between different behavioral descriptions (*e.g.*, signal, boolean, and arithmetic descriptions) of the same device, and a formalization of the relationship between the structure, behavior, and context of a device. Given these formal structures, my methods find sufficient conditions on the inputs of circuits which guarantee the correct operation of the circuit in the desired descriptive domain. These methods are algorithmic and complete. They also handle complex phenomena such as races and charge sharing. Informal notions such as races and hazards are shown to be derivable from the correctness conditions used by my methods.

Thesis Supervisor: Dr. Gerald Jay Sussman
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank the following people for helping me improve the presentation of the ideas in this dissertation: Jerry Sussman, Charles Leiserson, Miriam Leeser, Jerry Roylance, and especially Laura Yedwab.

To my friends and others I have known here, Phil Agre, John Batali, Alan Bawden, Howard Cannon, David Chapman, Gary Drescher, Carl Feynman, Ken Forbus, Margaret Fleck, Joe Halpern, Ken Haase, Danny Hillis, Dan Huttenlocher, Scott Jones, Tom Knight, Mike Greenwald, Chris Linblad, Neil Mayle, David McAllester, Albert Meyer, Henry Minsky, Margaret Minsky, Marvin Minsky, Victoria Pigman, Ron Pinter, Kent Pitman, Jonathan Rees, Bill Rozas, Howie Shrobe, Laurel Simmons, Richard Stallman, Jon Taft, David Wallace, Daniel Weld, Brian Williams, and John Wroclawski, it has been a pleasure knowing you all and thanks for making this such an interesting place.

I would like to thank Marilyn Pierce, godmother to all EECS graduate students, for protecting us from the ravages of an otherwise inhumane department.

Finally, I would like to thank the Fannie and John Hertz Foundation for supporting me with a graduate fellowship. The freedom their support gave me was very valuable.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense, under Office of Naval Research contract N00014-86-K-0180 and contract N00014-85-K-0124.

To the Memory of Dynamic Scoping

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This research is both about verification of synchronous MOS digital circuits, and about a particular program, called *Silica Pithecus*, which is an implementation of some of the ideas in this thesis. Verification of digital circuits is a new, not yet well understood field. Its development is critical to building larger provably correct circuits. The field is new enough that what verification is, what it means to verify a digital circuit, and how to verify a digital circuit are all largely unanswered (and sometimes unasked) questions. The goal of this research is to answer these questions. Among the issues that must be addressed are: describing digital systems, choosing appropriate electrical models, understanding how verification affects the design process, relating simulation and verification, understanding the role of abstraction, and relating structure, context, and function.

This dissertation concerns *multilevel verification* and *signal behavior generation*. Circuit behavior can be described at different levels of abstraction ranging from the analog level to the functional level (Figure 1.1). Multilevel verification proves that two different levels of behavioral description describe the same behavior. The higher (*i.e.*, more abstract) level description of a design's behavior is called its *abstract behavior*, the lower level description of a design's behavior is called its *concrete behavior*. The proof has two parts: abstracting the concrete behavior to the same domain as the abstract behavior, and then proving the abstracted behavior and the abstract behavioral description are equivalent.

The signal descriptive level and the digital descriptive level are the two descriptive levels we will primarily investigate. A signal is a time-varying voltage. The primitives operators of the signal level are transistors. A digital value is either 1, 0, or *float*. The primitive digital operators are `AND`, `OR`, `NOT`, and `join`. The digital descriptive level is time free and functional. It has no time delays or switching elements. This allows a digital behavior to be specified as simple combinations of primitives.[1]

[1] We will use the term "hardware digital level" to refer to a digital description which incorporates

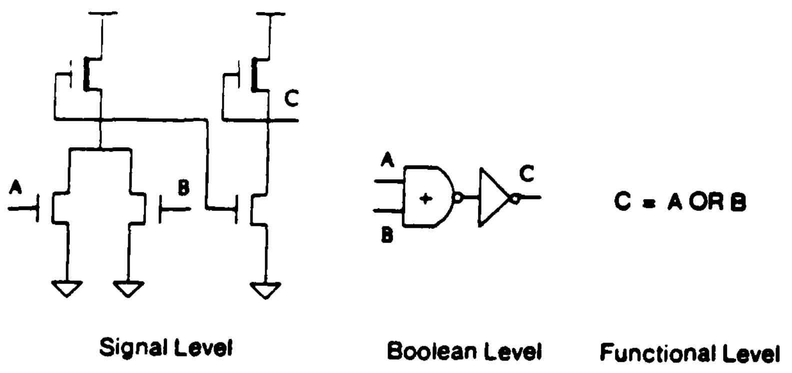


Figure 1.1: An example of multiple levels of behavioral description. Each descriptive level is an abstraction of the level to its left. There are many levels of abstraction besides the three shown here.

Property	Signal Level	Digital Level
Data Values	Signals	1, 0, *float*
Time Modelled?	Yes	No
Types of Primitives	Relational	Functional

Table 1.1: The signal level versus the digital level.

Digital and signal behavior differ in three ways (Table 1.1). First, digital values are incomparable to signal values. Second, circuit primitives are relational (*i.e.*, bidirectional) whereas digital devices are functional (*i.e.*, unidirectional). Third, time is important in circuits, the relative order in which events occur affects the final state of the circuit. Digital devices are time-free, they compute their results instantaneously.

The second focus of this dissertation is the automatic generation of a circuit's signal behavior from its structure (schematic). Signal behavior is the behavior of a circuit when viewed as a map from input signals to output signals. A signal is a time-varying voltage — intuitively, it what we observe on an oscilloscope when we probe a circuit. Generating the signal behavior of large circuits from their structure is one of the contributions of this research. The key is focusing the generation method on only those parts of a circuit's full behavior that account for the circuit's digital behavior.

Verifying circuits is difficult for many reasons (Table 1.2). First, circuits are huge. Chips of up to half a million transistors are becoming common. Brute force or linear techniques simply fail. Second, the signal behavior of a circuit must be derived

time and switching devices.

- Circuits are huge, up to half a million transistors.
- We must derive a circuit's behavior from its structure.
- We must employ accurate circuit models.
- A verifier must be fast.
- *Ad hoc* systems are not enough to prove correctness.

Table 1.2: Five reasons why verification is a hard problem.

- Exploiting the hierarchy and structure of schematics.
- Using *intentional analysis*.
- Employing circuit models accurate enough to model the most common bugs.
- Having a formal statement of circuit correctness.

Table 1.3: Four techniques for attacking the complexity of verification.

from the circuit's structure (schematic). Unfortunately, determining a circuit's full behavior is very hard and expensive. It is difficult to do either analytically or empirically. Third, accurate circuit models must be employed. Circuit models which are too simple will fail to catch bugs. On the other hand, circuit models which are too complex may make verification impossible. Fourth, an verifier must be efficient and fast to be usable. A designer needs quick response from his design system. Fifth, we want to prove our circuits are correct; merely finding and reporting bugs does not a verifier make.

I have designed and implemented a system, called *Silica Pithecus*, for the multi-level verification of circuits. It accepts the schematic of a circuit and a specification of the circuit's intended digital behavior and determines whether the circuit meets it specification. Silica Pithecus uses four techniques to solve the above problems (Table 1.3): it exploits the hierarchy of a design, it uses *intentional analysis*, it employs circuit models accurate enough to catch the most common bugs, and it has a formal statement of circuit correctness. These are discussed below, in turn.

The hierarchy that is used to create a complex circuit is exploited during analysis. Verification operates from the leaf nodes of a design to the root of the design.

All subcircuits of a circuit are verified before the circuit itself is verified. Two benefits accrue from this strategy. First, a circuit is verified only once — the first time it is used — and assuming it's always used for the same purpose, it need never be verified again. Second, verification can be performed incrementally as a design is created. The amount of work in verifying the composition of two previously verified subcircuits is proportional to the amount of new information created by the composition. The amount of work is largely unrelated to the complexity of the subcircuits themselves. In terms of a usable, interactive system, finding errors quickly and incrementally is extremely important. Detecting bugs as they occur, rather than waiting until a design is complete, causes reevaluation of designs at an early stage. It is not inconceivable that a design flaw in a small segment of a design invalidates the rest of the design. Detecting the flaw only after the design is completed is wasteful.

This approach keeps the design process interactive and incremental. Silica Pithecus is *interactive* because it can verify on demand any portion of the circuit, and it is *incremental* because it can analyze and verify parts of the circuit as they are entered. A designer is able to analyze and verify designs as they are created, rather than wait until the entire design (chip) is finished.

Intentional analysis is proving only the important properties of a circuit hold. The entire behavior of a circuit need not be known to verify that the circuit meets its digital specification. For example, it usually doesn't matter whether a signal's voltage at some time is 4.5 or 4.6 volts. All that is important is that it is above some threshold. Similarly, all that might matter about a signal could be its value at steady state. In this case it is unnecessary to calculate its intermediate values. By focusing on the relevant parts of a circuit's behavior and ignoring the irrelevant parts, predicting a circuit's behavior becomes a tenable proposition.

Silica Pithecus's component models are accurate enough to model (the sources of) threshold bugs, ratio bugs, charge sharing bugs, hazards, and races. These are the most common circuit bugs and the most complete set of bugs modelled by any one system (*cf.* Table 1.6). Silica Pithecus's models do not include capacitive coupling or unclocked feedback, so bugs arising from those phenomenon are not caught.

A formal statement of circuit correctness prevents Silica Pithecus from being *ad hoc*. The basis of the statement lies with the abstraction function used to map signals into digital values. The correctness statement guides much of the implementation, dictates the relationship between the actual and intended behaviors of a circuit, and formalizes the relationship between the structure, function, and context of a circuit.

My theory of verification is designed for verifying synchronous MOS VLSI digital circuits. We are therefore not concerned with asynchronous circuits or process technologies other than MOS. Neither are we concerned with non-digital designs, but it is harder to characterize what these are. For example, why is a bootstrap driver or a sense amplifier not a digital circuit? This dissertation defines as non-digital any circuit which relies on a race condition that can not be resolved using gate-delays. A bootstrap driver relies on a race between the driven node and the load node, it is non-digital. Similarly, a sense amplifier relies on a race that is not the result of gate delays.

Randy Bryant [Bryant86] imported the terms *false negative* and *false positive* into the vernacular of hardware verification. A false positive occurs when a verifier declares an incorrect circuit to be correct. Silica Pithecus is guaranteed never to give false positive. A false negative occurs when a verifier declares a correct circuit to be incorrect. Due to Silica Pithecus's conservative nature, it will give many of these. False negatives mainly arise when a circuit is more complex than Silica Pithecus's proof procedures and constraint processors can handle. Very often the form its false negative is "I can't prove this circuit is correct" rather than "This circuit is incorrect."

This chapter has seven sections. The first section gives an overview of Silica Pithecus and gives an example scenario. The second section discusses hierarchical verification. The third section gives some results of running Silica Pithecus on different circuits. The types of circuits which Silica Pithecus can verify are presented in the fourth section. Section five compares this research to other methods for verifying circuits. The sixth section highlights the contributions of this research. The last section gives an annotated chapter listing of this dissertation.

1.1 Scenario

A schematic of Silica Pithecus appears in Figure 1.2. Silica Pithecus has three inputs and two outputs.[2] Two of the inputs, the schematic of the circuit to be verified and the circuit's intended digital behavior, were discussed above. The third input, *logical constraints*, are boolean expressions on the (digital abstractions of the) inputs of the circuit being verified. These boolean expressions are guaranteed to always be true.[3] For example, logical constraints might dictate that the two clocks ϕ_1 and ϕ_2 are mutually exclusive. Other examples are dictating that a certain input is always asserted, or that either of two inputs will always be asserted.[4]

[2]This is an idealized representation of Silica Pithecus's input/output behavior. Silica Pithecus runs in a interactive system where circuits and programs coexist in the same database. Silica Pithecus has a couple of implicit inputs which are not represented in this diagram.

[3]All logical constraints are verified to be true when a circuit is used as part of a larger circuit. Silica Pithecus does not believe everything the designer says.

[4]There are two types of logical constraints, *steady state logical constraints*, and *temporal logical constraints*. Steady state logical constraints make assertions about the boolean interpretation of

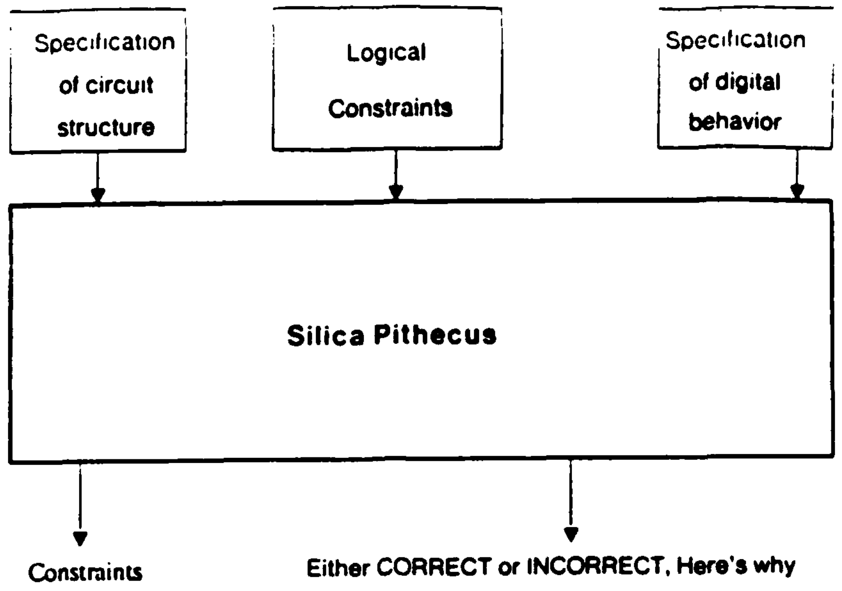

Specification of circuit structure
Logical Constraints
Specification of digital behavior
Silica Pithecus
Constraints
Either CORRECT or INCORRECT, Here's why

Figure 1.2: Silica Pithecus as it appears to the user.

One of the outputs of Silica Pithecus is a declaration of whether the circuit meets its digital specification, *i.e.*, is correct. If the circuit is not correct, then the reason it is not correct is given to the designer. Example reasons include reporting ratio bugs, threshold bugs, or glitches.

The other output is a set of constraints on the input signals. When satisfied, they guarantee that the circuit will exhibit its specified behavior. A given structure can exhibit many different behaviors; the constraints "project" out the desired subset of a structure's possible total behavior.[5] Structure alone does not determine function, structure and context together determine function. Examples of constraints include dictating that some input signal must not glitch or that some input signal must not suffer a threshold drop.

Constraints are a very powerful idea. They allow very large circuits to be verified. When a subcircuit's constraints are met, it means that the subcircuit will exhibit its specified digital behavior. Therefore, to generate the digital behavior of a circuit all of whose subcircuits have been verified, one need only prove all the subcircuit's constraints are met, and then compose the subcircuit's digital behav-

the values of signals when the circuit reaches steady state. Temporal logical constraints make assertions about the boolean interpretations of the values of signals for all time.

[5]The qualitative analysis community has a name for this observation, they call it the *No-function-in-structure* principle [de Kleer]. This principle states that an object's behavior is always predicated on the object's context.

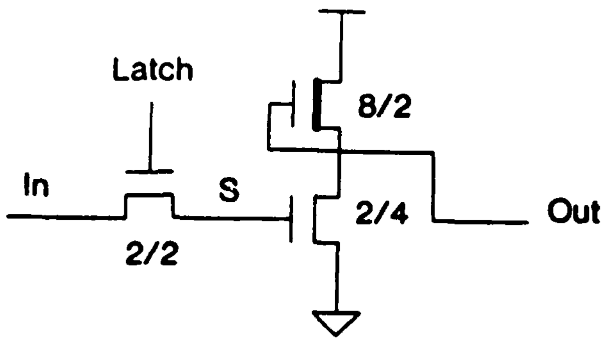


Figure 1.3: A Inverting Latch.

iors. This method of generating digital behavior is very fast because the signal level behavior of the circuit need never be generated. We will return to this topic when we discuss hierarchical verification.

Example

Consider an Inverting Latch (Figure 1.3). This device stores one bit on its S node. The output of the Inverting Latch is the negation of the value on its S node.

The digital specification of the Latched Inverter is

```
(df ((INVERTING-LATCH s) latch in)
  (stream-cons
      ; this is the output
      (if latch (not in) (not s))
      ; this is the next state
      (INVERTING-LATCH (if latch in s)))).
```

This program says that an Inverting Latch has one bit of state, s, and two inputs, latch and in. The inverter "returns" two values, the output and the next state of the device. The output and next state both depend on the state of latch.[6]

When Silica Pithecus is handed the schematic and digital specification of the Inverting Latch, it declares that the circuit is correct and generates four constraints (Table 1.4). The first constraint requires that whenever $\text{Latch}_\downarrow$ is true, signal flow must be from In to S. If signal flow is not from In to S, say, because In is a precharged bus and S has more capacitance than In, then the circuit will not have its intended digital behavior. The second constraint requires that if $\text{Latch}_\downarrow$ falls

[6]This is not a complete description of the program. A complete description appears in Chapter 7.

1. $\text{Latch}_b \Rightarrow$ `overpowers`($[\text{In}]_{In}$, $[\text{S}]_S$)
2. `falls`(Latch_b) $\Rightarrow$ `falls-first`(Latch_s,In_s)
3. `control`(Latch_s)
4. `no-drop`(Latch_s)

Table 1.4: Constraints on the latched inverter cell. Latch_s refers to the signal at node Latch. Latch_b refers refers to the boolean interpretation of the signal at node Latch. The other notations are described in later chapters.

during a computation[7] then it can only do so before In_s changes state. That is, In_s must either be stable on computations when Latch_s falls, or, if In_s is unstable when Latch_s falls, then In_s must change after Latch_s falls. The third constraint requires that once Latch_s is asserted, it must remain asserted. The fourth constraint requires that Latch_s not suffer a threshold drop. Latch_s is prevented from suffering a threshold drop so that S_s does not suffer two of them. If any of these constraints are violated, the circuit will not exhibit its intended digital behavior.

When a verified subcircuit is used as part of a larger circuit the subcircuit's constraints are checked. I anticipate that most errors in a design will be caught by the detection of violated constraints. Designers constantly work within semi-conscious digital abstractions of the analog properties of their designs. It is when they unknowingly break these semi-conscious digital abstractions that errors arise. The constraints that Silica Pithecus generates embody these semi-conscious digital abstractions, therefore breaking the now concrete abstractions cause the constraints to be violated, and an error to be returned to the designer.

1.2 Hierarchical Verification

Silica Pithecus spends most of its energy generating and abstracting a circuit's signal behavior. Hierarchical verification is a method of avoiding much of this work. Consider a circuit composed of previously verified subcircuits (Figure 1.4). When the digital behavior of the whole can be derived from the digital behavior of the parts the signal behavior of the whole device need not be generated.

This is precisely what hierarchical verification does. Hierarchical verification first processes the constraints of each subcircuit. If no constraint is violated, then the

[7] A *computation* starts when some clock changes state. A *computation ends* when the circuit reaches state. This idea is further elaborated in Chapter 3.

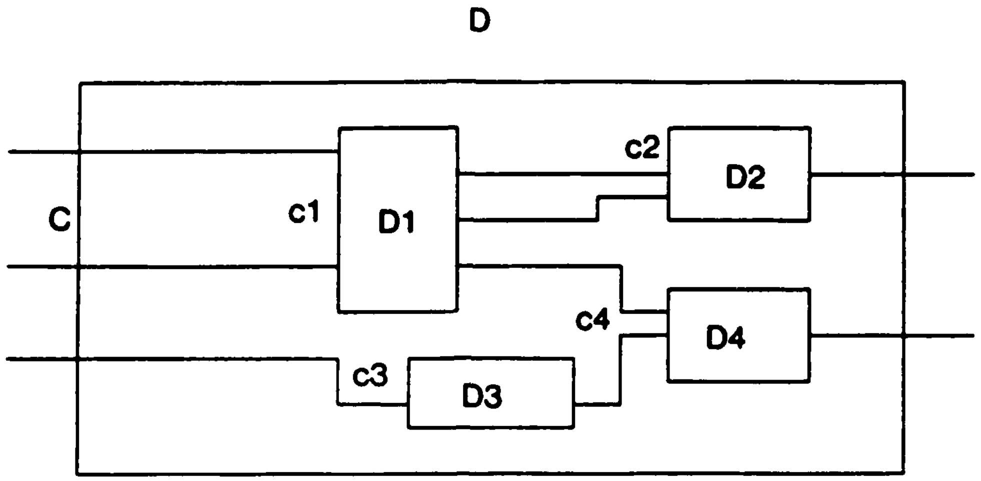


Figure 1.4: Hierarchical Verification. D is the whole design. Each part d_i is assumed to be verified. The c_i's represent the constraints on each d_i's inputs. Assuming that none of the constraints are violated, the digital behavior of the whole design can be derived by composing the parts' digital behaviors. A constraint that can be neither proven nor disproven is propagated as a constraint of the top level circuit D. C stands for the propagated constraints.

digital behavior of the whole is derived from the digital behavior of the subcircuits.

There are three possible outcomes of processing constraints: a constraint can be *accepted*, *rejected*, or *propagated*. A constraint is accepted when it is shown to hold. A constraint is rejected when is is proven not to hold. Rejected constraints are returned to the designer as errors. When there is not enough information to show that a constraint either holds or doesn't hold, the constraint is propagated up the structural hierarchy to be reprocessed when the circuit itself is used as a subcircuit. For example, because the input constraints C1 of component D1 in Figure 1.4 neither hold nor fail to hold, they are propagated as constraints on the inputs of D. Processing constraints is fast and efficient. There are five types of constraints. They are enumerated and discussed in Chapter 11.

As a concrete example of hierarchical verification, consider a Shift Cell built out of two Latched Inverter Cells (Figure 1 5). The specification of this device employs the specification of the Inverting Latch.

```
(df (SHIFT-CELL s1 s2)
  (let ((Left (INVERTING-LATCH s1))
        (Right (INVERTING-LATCH s2)))
```

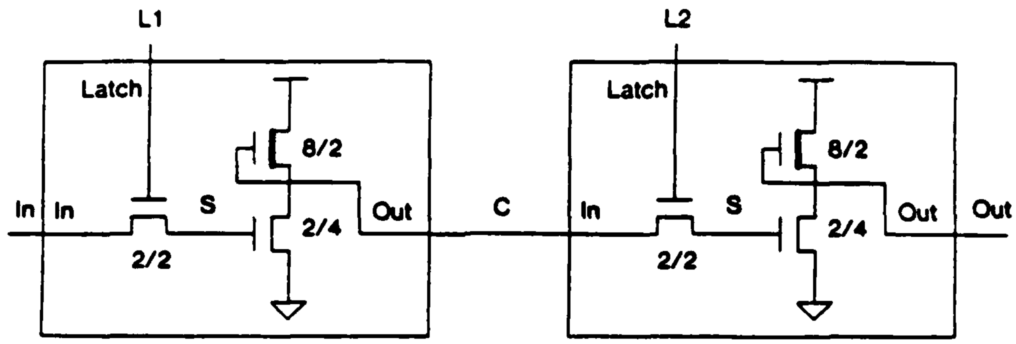


Figure 1.5: Composing Two Latched Inverter Cells to Make a Shift Cell

```
(lambda (in l1 l2)
  (Right (Left in l1) l2))))
```

This digital specification says that a Shift-Cell takes two inputs and that its behavior is equivalent to two inverting latches, one feeding the other.

There is one logical constraint for the shift cell: **tmutex**($L1_s$, $L2_s$). It declares that $L1_s$ and $L2_s$ are mutually exclusive.

When given the above information, Silica Pithecus declares the Shift Cell to be correct and generates the following six constraints:

- $L1_b \Rightarrow$ overpowers($[\mathrm{In}]_{In}$, $[(\mathrm{S\ Left})]_{(S\,Left)}$)
- control($L1_s$)
- control($L2_s$)
- no-drop($L1_s$)
- no-drop($L2_s$)
- **falls**($L1_b$) $\Rightarrow$ **falls-first**($L1_s$, In_s)

These constraints result from propagating the unresolved constraints of the two Inverting Latches to the Shift Cell. Of the eight constraints (four for each Inverting Latch), only two were satisfied:

- $L2_b \Rightarrow$ overpowers($[\mathrm{C}]_{C}$,$[(\mathrm{S\ Right})]_{(S\,Right)}$) and
- **falls**($L2_b$) $\rightarrow$ **falls-first**($L2_s$, C_s).

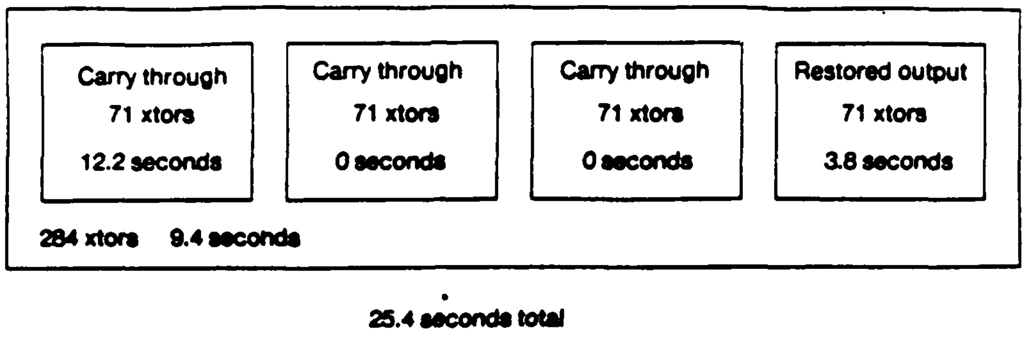


Figure 1.6: Verifying a 4 bit Manchester Adder.

Note that as part of propagation, the names of nodes are changed from being local to the Inverting Latches to being local to the Shift Cell. For example, no-drop(Latch,) (for the left cell) becomes no-drop(L1,). When a node has no corresponding name for it at the higher structural level a name based on the "pathname" of the node is generated for it. For example, the S node of Right is changed to (S Right) and the S node of Left is changed to (S Left).

1.3 Results

The time required to verify circuits is extremely encouraging.[8] Silica Pithecus has verified circuits of 9,000 transistors in 80 seconds. One particularly enlightening experiment is the one originally performed by Randy Bryant for MOSSYM [Bryant85] and repeated using Silica Pithecus.

In this experiment the ALU from [Mead/Conway] is programmed as an adder, which specializes it to be a precharged manchester adder [ref manchester adder]. This adder is a subtle device which uses many MOS features, and it is therefore an extremely good test case for an MOS verifier.

Silica Pithecus verifies a 16 bit adder (over 1100 transistors) built from this cell in 51 seconds. The breakdown of times for this is very instructive. Consider the verification of a four bit adder, which takes 25.4 seconds (Figure 1.6). The first adder cell takes 12.2 seconds to verify. To verify a four bit adder takes an additional 13.2 seconds. Of this additional time, 3.8 seconds is spent verifying a slightly different version of the adder cell where the carry-out line is restored, and 9.4 seconds processing the constraints of the four cells. By not reverifying the same part, and not generating signal behavior, hierarchical verification avoids extra work.

[8]The timings here are preliminary. Silica Pithecus is not completely implemented. The timings are all within a factor of two, probably a lot less. By no means is Silica Pithecus tuned. A tuned version might run much faster.

1.4 Other Work

Other work in ensuring circuits operate correctly is in four broad areas, bottom-up and top-down methodologies, simulation, electrical rules checking, and verification.

1.4.1 Verification Versus Methodology

Bottom-up and top-down methodologies are used to build circuits free of certain bugs.

Bottom-up methodologies have two components: restrictions on the types of circuits that can be built, and restrictions how circuits can be composed to make larger circuits. These restrictions eliminate both certain classes of bugs and circuit classes of circuits. For example, threshold bugs are one type of bug that can be prevented by requiring that all signals coming from pass logic be restored. The cost of preventing this particular bug, however, is very high. Requiring all signals to be restored excludes many types of circuits from being designed and costs extra silicon.

Top-down methodologies start with a high-level, and therefore presumably correct, specification of a circuit. The specification is incrementally refined into a concrete circuit. A fixed set of transformations are used during the refinement stage. Each transformation is guaranteed to preserve the correctness of the original specification and not to introduce any circuit bugs.

Both top-down and bottom-up methodologies throw out the baby with the bath water: they make inefficient use of chip area and create slow designs.[9] Furthermore, methodologies can't eliminate all classes of bugs — if they did, a designer would be overly restricted. For these reasons, we reject forcing designers to use a particular methodology.

1.4.2 Simulation

Today's designers debug their designs using simulation. Regardless of the level of simulation, simulation has many problems (Table 1.5), the foremost of which is simulation cannot guarantee the absence of bugs. As designs become more and more complex, simulation becomes less and less reliable. Exhaustive simulation must be performed to prove circuits are correct. But exhaustive simulation for large circuits is either exponentially expensive (when a boolean or digital domain is used) or impossible (when an infinite domain such as signals is used). Verification, which proves that a circuit works as intended for all inputs and state transitions, is completely reliable. Another problem is that simulation decontextualizes bugs.

[9]There is no theoretical reason for the resulting inefficiencies. Nonetheless, current technologies are wasteful of resources. Future research may alleviate this problem.

Simulation	Verification
Is not complete	Is complete
Decontextualizes bugs	Pinpoints bugs
Is non-compositional	Operates hierarchically
Weak circuit models	Accurate circuit models

Table 1.5: The Failure of Simulation. Simulation has many problems which verification does not have.

The output of a simulator is 1's and 0's. When a 0 appears where a 1 was expected (or vice versa, or when an unexpected X appears), there is no clue about the source of the bug (*i.e.*, whether it comes from a logic bug, ratio bug, *etc.*). The designer must laboriously track down the source of the bug. This is true even when exhaustive simulation is performed. A verifier automatically pinpoints the parts of an implementation which do not match their specification; this precisely locates bugs. A third problem is that simulation is non-compositional. Even when all of a circuit's subcircuits are verified via simulation, the circuit itself must be fully simulated. Non-composibility is the key reason simulation cannot be used to verify large circuits. Finally, because simulators must constantly reevaluate the same subnetworks over and over again, simulators must used impoverished circuit models. Accurate circuit models would be too expensive to use for large scale simulation. Because verification only looks at each possible network once, it can use a more elaborate circuit model.

Symbolic Simulation

Recently *symbolic simulation* has been proposed for the verification of digital MOS circuits. Symbolic simulation uses variables and constants during simulation [Bryant85]. When a circuit is symbolically simulated the result is an algebraic expression. This algebraic expression is then verified to be the expression expected. Due to symbolic simulation's algebraic structure, it is capable of simultaneously simulating a circuit for many different inputs. This feature makes exhaustive simulation easier for some circuits.

The major problems with symbolic simulation are lack of coverage, inability to catch races and hazards, decontextualization of bugs, and lack of composability. These problems arise because symbolic simulation is tied to a simulation paradigm thereby inheriting the problems of simulation.

Lack of coverage means that symbolic simulation does not prove that for all inputs and states a circuit gives the correct output and goes to the correct next state. The problem is that a purely symbolic simulation would simulate transitions and states that never occur and would yield many spurious errors. To avoid this problem, a designer must select simulation sequences that yield only states and transitions that can occur during real usage of the design. For example, if a device has two mutually exclusive inputs, then it is symbolically simulated twice: once with one input high and the other low, and once with the inputs reversed. Therefore, to get complete coverage, a designer must carefully craft a set of test sequences which fully exercises the chip.

MOSSYM does not use MOSSIM's unit delay model. Instead, it uses Näher's [Näher] *rank-order* delay model. Therefore MOSSYM cannot verify the timing behavior of circuits. Bryant advocates the use of ternary simulation [Bryant83] as an extra pass to get around this problem. However, ternary simulation only finds races, therefore this methodology cannot verify circuits which rely on races.

Like normal simulators, symbolic simulation decontextualizes bugs. The only difference is that symbolic simulation outputs boolean expression where normal simulation outputs ones and zeros. When discrepancies arise between the expected and actual results of simulation, the designer designer must still determine the cause of the discrepancy.

Symbolic simulation is not composible. It is a monolithic tool and cannot take advantage of hierarchy. If two subcircuits are verified via symbolic simulation, this gives no handle what happens when the two subcircuits are composed. Symbolic simulation is at least quadratic in its running time, therefore it cannot be used for large circuits.

Understanding why symbolic simulation is non-composable is very instructive. The basic problem is that the simulator assumes inputs are both driven and stable. For many circuits this assumption is invalid. The real inputs to a circuit may be precharged values rather than driven values, and they usually are not stable. Symbolically simulating a circuit does not indicate what the circuit does when connected to real signals. Therefore when a previously "verified" circuit is used as part of a larger circuit, the larger circuit must be simulated in its entirety.

1.4.3 Electrical Rules Checkers

Electrical rules checkers ensure that circuits are not malformed according to some structural criteria. Electrical rules checkers suffer from two major problems. First, they are completely *ad hoc* because they check that very specific configurations of circuits don't arise. Second, they often give many annoying spurious errors.

The first commonly used electrical rules checker was Baker's [Baker]. This checker made sure that each node could be potentially pulled up and pulled down, that depletion mode transistors were used in known ways, that two or more threshold drops didn't occur, that Vdd and Gnd weren't shorted together, and that proper ratios existed in gates. Its major problem was finding a tremendous number of spurious errors because it had absolutely no understanding of a circuit's logical structure (*e.g.*, which signals are mutually exclusive). It also had the problem of being non-hierarchical, it would find the same bugs over and over again. The output was so unwieldy that programs were written to filter out some of the spurious errors and to collect the repeated errors together.

TV [Jouppi] was a timing analyzer that used some knowledge of a circuit's operation to avoid the problems of spurious errors. It applied a set of *ad hoc* rules over a circuit to determine the direction information flowed through transistors. An electrical rules checker built on top of TV which had no spurious errors when run on sample circuits. One must use such a tool with caution, however. It suffers from two related problems. The first is that TV's methods for determining information are purely heuristic and can fail without the designer knowing. Second, TV assumes that ratios are correct to determine information flow, it then uses information flow to check ratios. This approach can fail in expected, and undetected, ways.

Karplus [Karplus] also has an *ad hoc* approach to electrical rules checking. He treats circuits as graphs and then uses graph algorithms to prove certain paths (sub-circuits) don't occur in the circuit being checked. Examples of the graph formation rules he checks are

1. Avoid shorting power.
2. Avoid changing the inputs.
3. Avoid parallel pullups.
4. Avoid gates in the middle of pulldown chains.
5. Avoid charge-sharing.
6. Avoid odd inverter cycles.

Some of these rules are not well-motivated and potentially unneeded. For example, Rule 2, avoiding changing the inputs, is not needed when the input is connected to the output of a gate. The reason for avoiding parallel pullups is to make ratio checks less prone to error. The reason for Rule 4, avoiding gates in the middle of pulldown chains, is so that *ad hoc* programs which try to determine signal flow won't get confused. Having *ad hoc* rules so that *ad hoc* programs will work better seems like poor methodology. Also, enumerating the ways a circuit can be malformed is not a good strategy. It is not clear whether all bad circuit formations can be enumerated.

1.4.4 Other Types of Verification and Other Verifiers

Very little work similar to the work discussed in this dissertation has been done. People have worked on multilevel verification, and have worked on generating behavior from structure, but always with a different focus than that presented here.

The technologies and models used by previous researchers differ from mine. Specifically, earlier work on deriving behavior from structure were either for TTL [Wagner], or used inadequate circuit models for MOS [Gordon, Barrow82]. The types of circuits built from TTL and from MOS are completely different, and the models of [Gordon82] and [Barrow] can't catch charge-sharing bugs, ratio bugs, threshold bugs, races, or hazards. Also, Wagner's verifier was not automated and could only handle small circuits. Barrow's verifier is discussed in detail below.

Gordon has since improved his model to make transistors be bidirectional and to include time [Gordon84b]. He uses higher order logics for describing and verifying circuits [Gordon84]. Using logic is an improvement over more functional approaches. Logic easily captures the bidirectionality of transistors and logic is well known. However, as used, it is no better than earlier formalisms for describing, or reasoning about, the electrical aspects of circuits.

Research on multilevel verification has also been done [Eveking] [Gordon84c]. Previous work made explicit the notion of an abstraction function for mapping between levels. Some even had partial abstraction functions. However, none have the fully understood the role of constraints. One researcher [Eveking] noticed that constraints (he called them assertions) were needed to ensure inputs were valid (*i.e.*, mapped into the higher level descriptive domain). He, however, did not make the observation that constraints were also needed to ensure outputs were valid. This observation is one of the contributions of this dissertation and is the basis for automatically generating constraints.

Other attempts at verifying MOS circuits will now be discusses, followed by an outline of other work in verifying diverse properties of digital systems.

Verify

Besides Silica Pithecus, the only other implemented verifier for MOS circuits is Verify [Barrow]. Verify differs from Silica Pithecus in two very important ways (Table 1.6). First, it has no explicit notion of abstraction or multiple levels of description, it is a *monoverifier*. Second, its circuit model is extremely weak. It cannot model charge-sharing bugs, ratio bugs, threshold bugs, races, or hazards.

Verify's focuses on verifying digital designs. It models circuit components and networks of components as time-free digital devices. (That is, Verify assumes the result that Silica Pithecus is trying to prove, namely, that circuits can be described as time-free digital devices.) Given two different digital descriptions of the same design, it can determine whether the two descriptions are the same. One of the descriptions is derived directly from the structure of the design, the other is the specification. Because the circuit's behavior and its specification are described at the same level (*i.e.*, the digital level) an abstraction function is unneeded.

(Silica Pithecus could use the capabilities of Verify. When Silica Pithecus verifies

Researcher	Program	Domain	Transistor Model	Wire Model	Bugs Modelled
Barrow	Verify	Tristate	Unidirectional (Functional)	1 Bit	Logic
Gordon		Tristate	Relational	1 Bit	Logic Timing
Bryant	MOSSIM MOSSYM	Many Valued Signals	Bidirectional Switch	Capacitors	Charge Sharing Timing Logic
Terman	RSIM	Interval Signals	Varying Resistors	Capacitors	Charge Sharing, Ratio bugs, Timing Logic
Weise	Silica Pithecus	Signals	Varying Resistors + Threshold Device	Capacitors	Charge Sharing, Ratio bugs, Timing, Threshold Logic

Table 1.6: Verifiers and Simulators for MOS Circuits. This table shows the different characteristics of MOS verifiers and simulators. The top row highlights Verify. This verifier's strength is in verifying digital systems rather than verifying circuits. The second row highlights two popular simulators. These simulators have reasonable circuit models, but don't perform verification. The last row shows Silica Pithecus, which has the best of both worlds. It can formally prove a circuit is correct, and its models are better than the simulators' models.

large circuits, their structural and functional specifications must match. This makes comparing the abstracted and specified behaviors simple: Silica Pithecus shows that the subcircuits correspond to subfunction in the digital specification and that the wires in the circuit correspond to dataflow in the digital specification. The comparison of arbitrary digital expressions occurs only for leaf cells. Silica Pithecus and Verify are complementary programs, the former verifies circuits and the latter verifies digital designs.)

Functional Verifiers

Functional verifiers are multilevel verifiers where the concrete domain is the digital level and the abstract domain is at some higher level such as the arithmetic level. Functional verifiers are complex because the abstract domain are very rich. Proving that two expressions denote the same value can require difficult theorem proving.

An interesting functional verifier was created by [Hunt]. He began with the Boyer/Moore theorem prover [Boyer]. Wherever the theorem prover was inadequate it was extended by Moore. The primitives of Hunt's circuits were idealized boolean (strictly bistate) gates. He designed a microprocessor and verified that it met its specification. Because he employed the Boyer/Moore theorem prover his proofs were able to use induction. Therefore he could prove that parameterized descriptions of his circuits were correct, something no other verification system has done. For example, he was able to prove that his specification for an n-bit adder was correct regardless of its particular size. Other verifiers (*e.g.*, Silica Pithecus and Verify) allow parameterized descriptions, but the descriptions must be concretely instantiated before verification can be performed.

Protocol Verifiers

Protocol verifiers show that some communication and arbitration scheme properly controls some system. For example, protocol verifiers prove that a system will always service or acknowledge a given request. Protocol verifiers are also used to verify asynchronous systems. Protocol verifiers assume that primitives respond to changes in their inputs with some finite delay. The trick is to prove that regardless of the delays, which can't be accurately predicted, the system will still work.

Temporal logics are often used to specify the desired properties of asynchronous systems and communication protocols [Mishra/Clarke] [Moszkowski] [Fujita]. They formalize notions such as "before," "after," and "following."

There is some confusion in the community between verifying asynchronous systems and verifying asynchronous circuits. The problem is that an asynchronous circuit is an instance of an asynchronous system. Therefore researchers sometimes claim they are verifying circuits when their methods have little to do with circuits (other than circuits are used to implement asynchronous systems). Their claim is partially true, however, the claim is beside the point. One could implement an asynchronous system using people, instead of circuits, and their methods would still apply.

Verification of Sequential Systems

This category of verifiers proves that a system exhibits the correct behavior over many cycles. Examples include verifying that a bit serial adder really adds two numbers, or verifying that microcode correctly implements its intended macrobehavior.

Some indirect work has been done, in an *ad hoc* fashion on this topic. Both Verify and MOSSYM have been used to verify a the correct behavior of a design over many cycles. Verify was extended to verify, with human intervention, the microcode of a small computer. MOSSYM has verified bit serial adders.

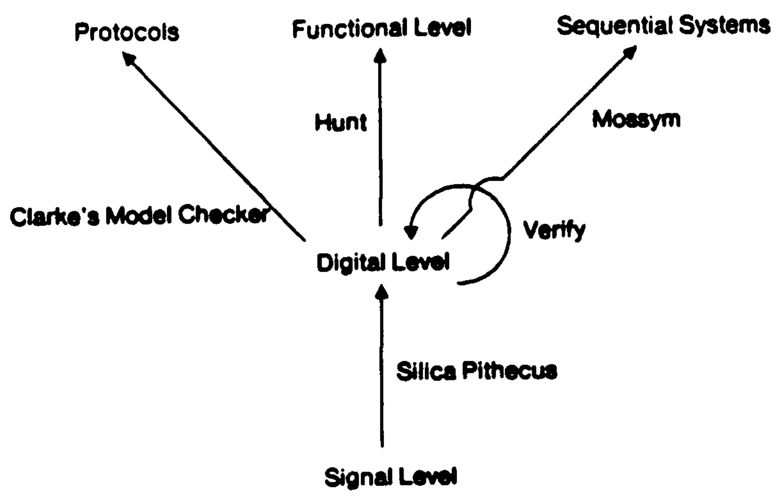


Figure 1.7: Silica Pithecus's place in the verifier community.

The only direct work that has been done on this is in the field of microcode verification.

Silica Pithecus's place in the verifier community is shown in Figure 1.7.

1.5 Contributions of this Research

The main contribution of this research is a formal abstraction mechanism for verifying the signals of a circuit correctly implement the digital abstractions envisioned by the designer. This includes, for example, proving signals can be viewed as ones and zeros, that nodes are capable of storing bits, and that control signals don't glitch. A major difficulty is deciding which signals are intended by the designer to be state bits, control signals, or data signals. These concepts are related to the role of a signal and are unrelated to electrical considerations.

The second contribution is an efficient method for the automatic generation of a circuit's signal behavior from its structure (schematic). This is one of the contributions of this dissertation. The key is focusing the generation method on only those parts of a circuit's full behavior that account for the circuit's digital behavior. No previous system with reasonable circuit models has predicted the behavior of large MOS circuits solely from their structure.

Other contributions and ideas are listed below.

A formal model of multilevel verification.

I precisely define what it means to prove a circuit is correct. Other researchers in multilevel verification have had a similar statement of correctness, but they all missed the role of *context* in their definitions of correctness.

A method for formally deriving constraints.

Besides specializing behavior, the purpose of constraints is ensure that the behavior of a given design is abstractable to the next higher level of description. This observation yields a method for automatically generating many of the constraints on a design.

A formal statement of the relationship between structure, function, and context.

Hierarchical verification depends on the explicit dependence between structure, function, and context.

A method for symbolically predicting the behavior of a circuit solely from its initial state and inputs for any valid state and inputs.

As mentioned several times in this chapter, Silica Pithecus must both derive a circuit's signal behavior from the circuit's schematic and also abstract the signal behavior to the digital domain. A major contribution of this thesis a method for efficiently deriving a circuit's signal behavior from its schematic.

Intentional analysis.

Intentional analysis automatically falls out of the statement of circuit correctness. It allows a verifier to concentrate solely on the behavior of a design that contributes to the design's behavior at the higher levels of abstraction.

A Constraint System.

The constraint system serves a dual role. The constraints themselves serve as part of the proof that a given circuit will exhibit its intended digital behavior. Their other role is the mechanism by which very large circuits are verified.

Accurate models.

Verification would be meaningless if there were large classes of bugs which could not be modelled. This research uses component models capable of modeling common circuit bugs.

An existence proof that the theory works.

Silica Pithecus is proof that a verifier with an accurate circuit model can be made to run, and run quickly, for large circuits.

1.6 Organization of the Thesis

This dissertation has four major chunks: the circuit model (Chapter 2), the theory of multilevel verification (Chapters 3 through 5), the structure of Silica Pithecus (Chapters 6 through 9), and the theory and practice of hierarchical verification (Chapters 10 and 11). An annotated chapter listing appears below.

Chapter 1, Introduction

This chapter delineates issues, presents an example, introduces terminology, summarizes results, and presents an annotated chapter listing.

Chapter 2, Component, Signal, and Network Models

Chapter 2 presents Silica Pithecus's component, signal, and network model. The signal model describes the primitive elements manipulated by circuits. The component model describes the primitive devices for manipulating signals. The network model describes what the meaning of interconnected components is. An analogy can be made between circuits and programming languages, where signals correspond to primitive data types, components correspond to primitive operations, and programs.

Chapter 3, Multilevel Verification

There are many different levels of description (*e.g.*, signal, digital, functional) of a circuit's behavior. Multilevel verification is defined as proving that two different levels of description describe the same behavior. The lower level of behavioral description of a circuit's behavior is the circuit's concrete behavior, the higher level description is its abstract behavior. Chapter 3 presents the formal basis of multilevel verification. It presents examples of both of verifying a circuit's signal behavior against its digital behavior, and of verifying a circuit's digital behavior against its functional behavior.

Chapter 4, Time and State

Chapter 4 discusses how state and time are incorporated into the formal model of the previous chapter. The major result of Chapter 4 is method for symbolically predicting the final state of a circuit solely from its initial state and inputs. The prediction does not perform a simulation of any kind.

Chapter 5, Non-electrically Isolated Inputs

Chapter 5 extends the theory of Chapters 3 and 4 to include circuits with non-electrically isolated inputs.

Chapter 6, Silica Pithecus

An overview of Silica Pithecus is presented in Chapter 6. The major parts of Silica Pithecus are discussed in the next four chapters. Parts of Silica Pithecus which do not deserve their own chapters appear in Chapter 6. This includes the representation of circuits and Silica Pithecus's comparison methods.

Chapter 7, The Specification of Digital Systems

The issues in specifying both digital behavior and digital systems is discussed. A language for the specification of digital systems is presented. This language has an interpreter so specifications can be debugged by executing them.

Chapter 8, Generating Net Behavior

Net behavior is a structural representation of a circuit that emphasizes the different nets that a node might be a member of during settling. Chapter 8 presents an algorithm for generating net behavior.

Chapter 9, From Net Behavior to Digital Behavior

Rather than generate signal behavior from net behavior and then abstract signal behavior to the digital level, it is more efficient to generate digital behavior directly. Chapter 9 discusses how the network model is invoked to interpret nets and how the interpretation is abstracted to the digital domain.

Chapter 10, Hierarchical Verification

The theory and practice of hierarchical verification and hierarchical multilevel verification is presented in Chapter 10.

Chapter 11, Processing Constraints.

Silica Pithecus uses five types of constraints. Each of these constraints and how they are processed is presented in Chapter 11. The preliminary processing of constraints is identical, but each type of constraint has it own special theorem provers.

Chapter 12, Future Research

Chapter 12 discusses problems I never got around to solving. This includes using the information needed for verification to create an accurate timing estimator, handling unclocked feedback, dealing with capacitive coupling, and multiple abstraction functions.

Chapter 2

Component and Signal Models

We now turn our attention to the signal, component, and network models used by Silica Pithecus. This modularization corresponds to describing the semantics of a programming language in terms of its primitive data types, primitive operations on these types, and what the meaning of a combination of primitive operations (called a program) is.

The component model used by Silica Pithecus is as accurate as the component model used by any large scale simulator.[1] Silica Pithecus's component model is superior to MOSSIM's model, and, not counting modeling how long a circuit takes to settle, is superior to RSIM's model. To a first approximation, if MOSSIM or RSIM can simulate a circuit, then Silica Pithecus can verify it.[2]

Silica Pithecus could employ a simpler model than it does, such as MOSSIM's switch level model. This would save some analysis time. However, the time savings would not be worth the less accurate verification that would result. For example, Silica Pithecus would not be able to detect threshold or ratio bugs if it adopted MOSSIM's model.

This chapter has five sections. The first section describes Silica Pithecus's component model. A new way of describing circuit a circuit's dynamic structure, called *net behavior*, is presented in the second section. The third section describes Silica Pithecus's network model, that is, what interconnected components mean. Silica Pithecus's signal model is described in the fourth section. A discussion of the bugs which can and cannot be modelled are presented in the fifth section.

[1] A large scale simulator is a simulator that handles circuits in excess of 10,000 transistors.

[2] This statement is only partially true, as Silica Pithecus won't verify circuits that use unclocked feedback and it may not be able to prove some correct circuits are correct, but for most designs it is true.

2.1 The Component Model

Silica Pithecus uses a component and network model similar to RSIM's model [Terman] except for RSIM's features which are concerned solely with timing. For example, there is only one effective resistance for any given transistor, rather than three. We will first describe transistors, then nodes.

Transistors

Conducting transistors are modelled as resistors in series with a *threshold drop device*. The resistance of a transistor[3] depends on the gate voltage of the transistor and the channel length and width of the transistor. A threshold drop device appears in circuit diagrams as a square with an X in it.

The resistance depends on three factors:

1. **Length and Width:** The resistance of a transistor is proportional to its channel length divided by its channel width.

2. **Type:** The type of a transistor determines the resistance of a unit square of the active region.

3. **Gate Voltage:** The resistance is doubled if the gate voltage has suffered a threshold drop.

Silica Pithecus's model for an enhancement mode transistor divides a transistor into four operating regions (Figure 2.1). The X'd square represents a threshold drop device. This device acts as a regulator preventing more than a certain amount of charge (electrons) from being pushed through it. For example, in the third case of the figure only enough charge can be pushed through it to raise the voltage on the node to its right to 3.5 volts. When the gate voltage is between 1.5 volts and 3.5 volts the resistance of the transistor is unknown (denoted X). A depletion transistor is modelled as a resistor of resistance L/W with no concomitant threshold drop device.

This is a simple transistor model. There are many real phenomena it ignores or simplifies. Nonetheless, the model is accurate enough for our needs.

There is nothing sacred about our choice of voltages. We have chosen the upper rail as 5 volts and the lower rail as 0 volts for convenience. These voltages could have been left symbolic (we will sometimes write them as Vdd and Gnd, respectively), however, this wouldn't have change the results or theories of this dissertation.

[3]More accurately: "the resistance of the resistor the transistor is modelled as."

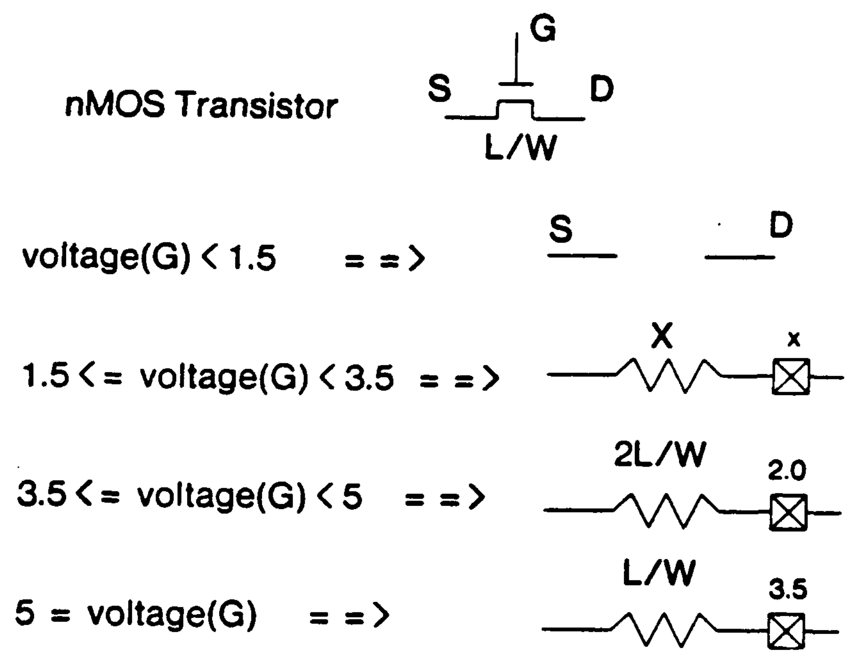


Figure 2.1: Silica Pithecus's Transistor Model. Enhancement transistors are modelled as resistors in series with a voltage controlled switch. The resistance of the transistor depends on the gate voltage. When the gate voltage is low enough then the transistor is modelled as an open switch. When the gate voltage is "full strength" the resistance of a transistor is L/W where L is the channel length and W is the channel width. The resistance is doubled when the gate voltage has suffered a threshold drop. The X'd square represents a threshold drop device. This device acts as a regulator preventing more than a certain amount of charge (electrons) from being pushed through it. For example, in the third case above only enough charge can be pushed through it to raise the voltage on the node to its right to 3.5 volts. When the gate voltage is between 1.5 volts and 3.5 volts the resistance of the transistor is unknown (denoted X). Depletion transistors are modelled as resistors of resistance L/W with no concomitant threshold drop device.

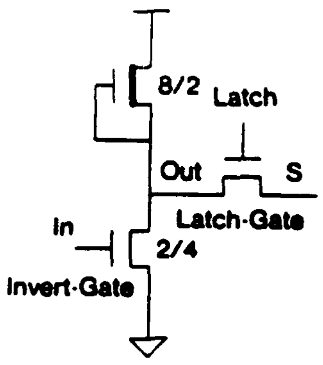


Figure 2.2: A Latched Inverter Cell. This cell differs from an Inverting Latch by the placement of the latch gate.

Nodes

Silica Pithecus models nodes as capacitors to Ground. Capacitance is measured in dimensionless units. (When timing is a concern, capacitance needs proper units.) Nodes whose capacitance is not declared have a capacitance of 1.

2.2 Net Behavior

During settling a node is a member of many different nets. The different nets are called the node's *possible nets*. Depending on the pattern of conducting transistors and the topology of the circuit, the sizes of a node's possible nets can range from containing a single node (the node itself), to containing every node in the circuit. A node's possible nets and the conditions under which they are active is called the node's *net behavior*. Net behavior is a function that maps time into nets. Given some time t, the net behavior of node N returns the net that N is a member of at t. A circuit's net behavior is the sum of its node's net behaviors.

For example, consider a Latched Inverter Cell (Figure 2.2).[4] The S node of this device has three possible nets, which are shown in Figure 2.3. The net on the left, the single node S, arises when Latch_b is not asserted. The middle net occurs when Latch_b is asserted and In_b is not asserted. The net on the left arises when both Latch_b and In_b are asserted.

We will use a textual notation for nets: if a net contains nodes A, B, and C, then it is textually written [A, B, C]. For example, the abbreviations of the nets of Figure 2.3 are [S], [S, Vdd, Out], and [S, Vdd, Out, Gnd], respectively.

[4] A Latched Inverter Cell differs from an Inverting Latch (Figure 1.3) by the placement of the latch gate.

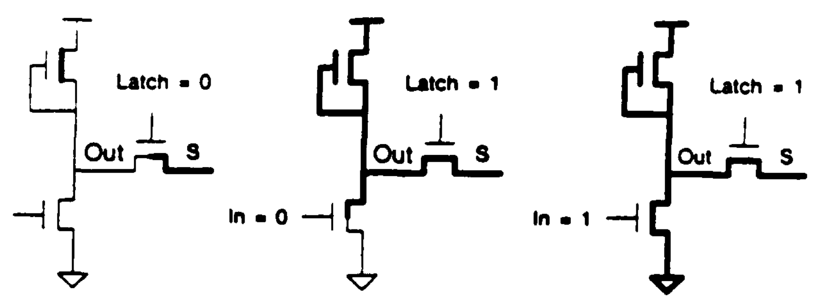


Figure 2.3: Possible Nets of the S Node of the Latched Inverter Cell.

Current doesn't flow between nets. This has two intertwined benefits. First, net behavior is a wedge by which we can modularize a circuit's behavior to make analysis possible. Second, all the possible sources of change to a node's voltage are contained in the node's possible nets. Therefore a node's voltage over time can be predicted by analyzing the node's net behavior. (This analysis is presented in Chapter 4.)

2.2.1 Representing Net Behavior

A node N's net behavior is represented by a *net behavior expression* written as McCarthy conditional [Lisp 1.5]. The conditional consists of a series of *clauses*. Each clause consists of one predicate and one net. The predicate is a boolean expression. If a predicate is true then N is a member of the predicate's associated net. N_{net} denotes N's net behavior.

For example, consider again the S node of the Latched Inverter Cell (Figure 2.2). The representation of its net behavior is

$$\begin{aligned} S_{net} = \lambda\, t\, .\ & \mathrm{Latch}_b(t) \wedge \mathrm{In}_b(t) \rightarrow [\mathrm{S\ Vdd\ Out\ Gnd}], \\ & \mathrm{Latch}_b(t) \wedge \mathtt{NOT}(\mathrm{In}_b(t)) \rightarrow [\mathrm{S\ Vdd\ Out}], \\ & \mathtt{NOT}(\mathrm{Latch}_b(t)) \rightarrow [\mathrm{S}] \end{aligned}$$

This formula is called a *net behavior equation*. The left hand side of a net behavior equation is the name of a node subscripted by *net* and the right hand side is a net behavior expression.

The net behavior of a circuit is a listing of the behavior equations of its nodes.

$$\begin{aligned}
\text{source}_{net} &= \lambda\ t\ .\ \text{gate}_b(t) \rightarrow [\text{source drain}],\\
&\qquad \texttt{NOT}(\text{gate}_b(t)) \rightarrow [\text{source}]\\
\text{drain}_{net} &= \lambda\ t\ .\ \text{gate}_b(t) \rightarrow [\text{source drain}],\\
&\qquad \texttt{NOT}(\text{gate}_b(t)) \rightarrow [\text{drain}]\\
\text{gate}_{net} &= \lambda\ t\ .\ [\text{gate}]
\end{aligned}$$

Table 2.1: Net Behavior Equations of the nMOS Transistor

For example, the net behavior of an n-channel transistor has three equations, one for each node (Table 2.1).

Efficiently finding the possible nets in a circuit requires understanding the *logical structure* of a design, *i.e.*, which signals are mutually exclusive and how some signals depend on other signals. Without this knowledge a circuit can appear to have an exponential number of nets with respect the number of transistors in the circuit, rather than the linear or quadratic number of nets intended by the designer. For example, a barrel shifter's control signals are mutually exclusive. Therefore, it has only a linear number of nets. But unless this knowledge is used, the barrel shifter will appear to have an exponential number of nets. Chapter 8 discusses the methods of employing the logical structure of a design during circuit analysis.

We now introduce a some terminology that is used throughout this dissertation. Off-chip inputs (*e.g.*, Vdd, Gnd, clocks) are assumed to be driven and are called *driven nodes*. Nets containing at least one driven node are called *driven nets*. All other nodes and nets are called *undriven*. What others have called a *storage node* we treat as a degenerate case of an undriven net containing only one node. A net *contracts* when some transistor in it shuts off thereby partitioning the net into two smaller nets. A net *expands* when one of its boundary transistor turns on thereby connecting it with some other net.

2.3 The Signal Model

Silica Pithecus models signals as functions which map time into a voltage and a strength. That is,

$$\textit{Signal} : \textit{time} \rightarrow \textit{voltage} \times \textit{strength}.$$

Intuitively, a signal is what we see on our oscilloscope when we probe one of the nodes of the circuit and watch the circuit in operation. Signals have two components, voltage and strength. The voltage of a signal is well understood and will not be explained. Signal strength is explained below.

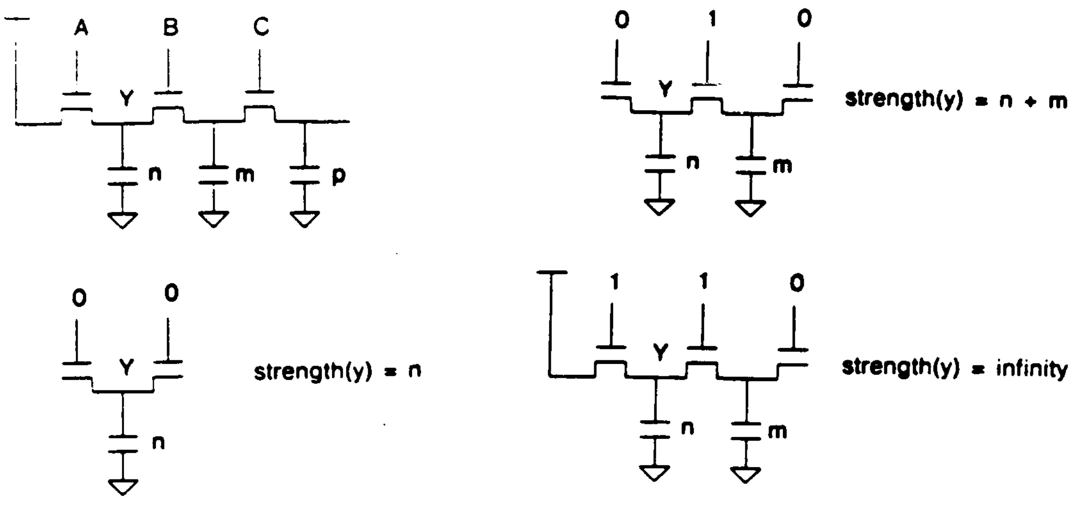


Figure 2.4: Signal Strength

2.3.1 Signal Strength

Signals emanate from nets of nodes (hereafter just called *nets*). The strength of a signal at some point in time is the the sum of the capacitances of the nodes in the net from which the signal emanates at that point in time. Driven nodes are considered to have infinite capacitance.

For example, consider the simple circuit of Figure 2.4. Assume that there is probe at node Y. We are going to measure the strength of Y_s, the signal at Y, for different states of the circuit. When all the transistors are off, the strength of Y_s is n. If B is then asserted, the strength of Y_s goes to $n+m$. Asserting A at this point causes Y_s's strength to be ∞ because the "capacitance" of VDD is ∞. A signal whose strength is ∞ is a called a *driven signal.*

The strength waveform of a signal[5] at a node encodes the charge sharing pattern of the node. An increase in X_s's strength indicates X is increasing the number of nodes with which it shares charge. A decrease in strength indicates that X is decreasing the number of nodes with which it is sharing charge. Knowing that a signals strength only decreases or only increases during a computation is very valuable. For example, when a signal's strength only decreases during a computation then the nodes in the signal's final net retain their initial charge. Chapter 4, which shows how a circuit's final state is predicted, discusses the strength of signals in

[5]A strength waveform is analogous to a voltage waveform. It is a plot of strength versus time.

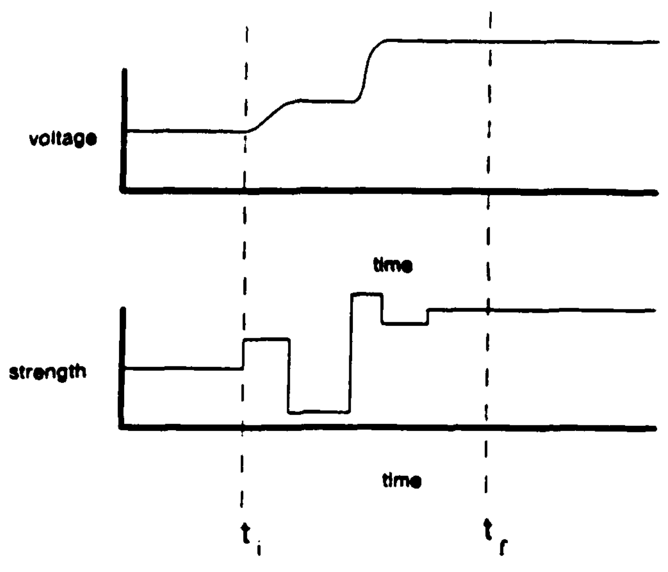


Figure 2.5: Notation for Time. We assume that a circuit is at steady state when its inputs change. By convention, t_i is the time at which the first input changes and t_f is the time when the circuit settles again.

detail.

2.3.2 Notation for Talking about Time

We assume that a circuit starts out at steady state, that its inputs change, and that the circuit then settles to a new steady state. We call the change of an input a *computation request* and that a circuit is caused to *compute* by a computation request. The time when the inputs first change is called the *initial time* and is denoted t_i. The *final time,* denoted, t_f, occurs when the circuit achieves steady state. These times are illustrated graphically in Figure 2.5.

As a consequent of this notation $X_s(t_i)$ denotes the value of X_s before a computation request is issued. $X_s(t_f)$ denotes the value of X_s when the circuit settles. A signal is *stable* during a computation request R (*i.e.*, from the time R starts to the time the circuit settles) if and only if the signal provably never changes during this time.

$N_{net}(t_i)$ is called N's *initial net.* $N_{net}(t_f)$ is called the N's *final* or *chosen net.* Following standard nomenclature, the predicates which select a node's initial and final nets are called the node's *precondition* and *postcondition*, respectively.

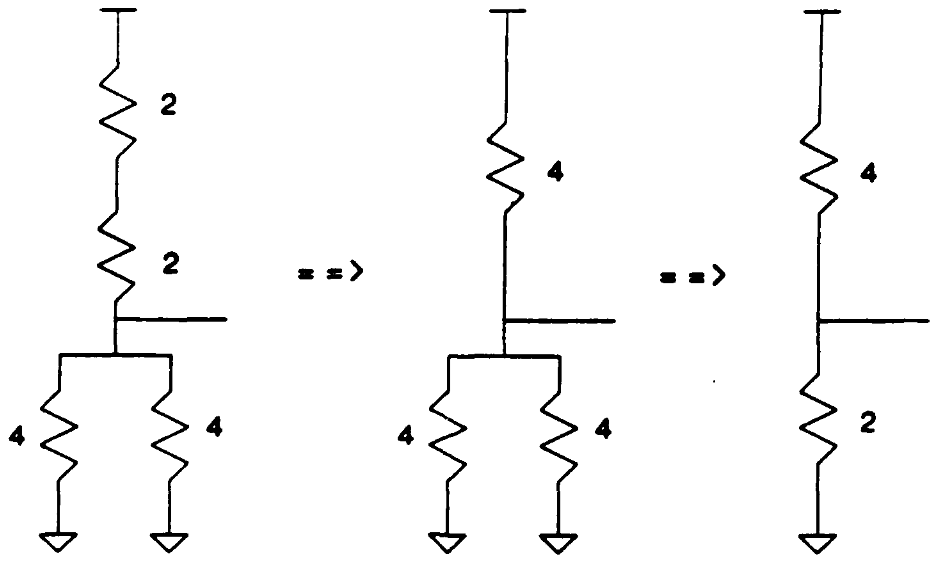


Figure 2.6: Creating Equivalent Voltage Dividers.

2.4 The Network Model

Given a net of transistors and a node of interest, *the network* model says how the net is interpreted to yield a voltage and a strength at the node. Thevenin equivalents are used to interpret net containing driven (*i.e.*, off-chip) nodes. A charge sharing model is used for nets not containing driven nodes.

The final voltage of a node N in a net containing driven nodes is determined by creating the Thevenin equivalent of the net regarding N as the net's output. The calculation is similar to the calculation presented by [Terman], except that intervals are not used in the calculation. The subnet "above" the node (*i.e.*, the subnet leading to Vdd) and the subnet "below" the node (*i.e.*, the subnet leading to Gnd) which form the voltage divider for the node are each reduced to yield characteristic voltages. It may be the case that there is no subnet "below" the node, in which case the node is pulled-up.

An example of creating an equivalent voltage divider appears in Figure 2.6. In this example a net with two transistors above the output node and two transistors below the output node are replaced by one equivalent transistor above the output node and one equivalent transistor below the output node.

Final voltages cannot be determined for nets not containing driven nodes because the verifier can't know the charge on the nodes in the net. Instead, Silica Pithecus locates a *dominant node* whose capacitance is sufficiently greater (more than 10 times) than the sum of the capacitances of all the other nodes in the net. If such a node exists, then all the nodes in the net will assume the voltage of the dominant node.

If such a node does not exist, then Silica Pithecus will signal a charge-sharing error. (Of course, Silica Pithecus is careful only to generate such errors for nets which must yield a valid voltage.)

2.5 Bugs Modelled and Not Modelled

We first discuss bugs that can be caught, then bugs that can't be caught.

2.5.1 Bugs Modelled

The bugs that can be modelled are listed below.

Charge Sharing A charge charging bug occurs when two or more nets share charge but neither has sufficient charge to "overpower" the other thereby resulting in invalid voltage levels. Circuit techniques such as precharged buses can be verified only by a system that properly models nodes. A verification system which modelled all nodes as having the same capacitance could not verify a circuit which used precharged nodes.

Ratio Bugs A gate in an nMOS circuit is a voltage divider. A gate is capable of driving its output to the upper rail, but incapable driving its output to the ground level. (Because that would imply that the pulldown has zero resistance.) The resistances of the two "halves" of the voltage divider must be carefully balanced to ensure that the output is close enough to ground to ensure a reliable zero level, but not so close that that the current flow through the gate is excessive. Ratio bugs can be modelled because transistors are modelled as resistors whose resistance is dependent upon the geometric characteristics of the transistors.

Threshold Bugs Transistors are not ideal switches. Transistors can only pass a certain voltage that is some fixed amount below the voltage on their gates. As signals pass through pass transistors they suffer threshold drops which is exhibited as a truncation of a signal's voltage. There are two different types of bugs caused by threshold drops. The first is ratio problems with gates because the resistance of a transistor depends upon its gate voltage (Figure 2.7). The second is threshold drops adding up when a signal suffering a threshold drop gates a transistor T causing the signal gated by T to suffer two threshold drops. Signals which suffer two threshold drops have lost enough voltage to become invalid signals. Threshold drops are modelled by including an explicit

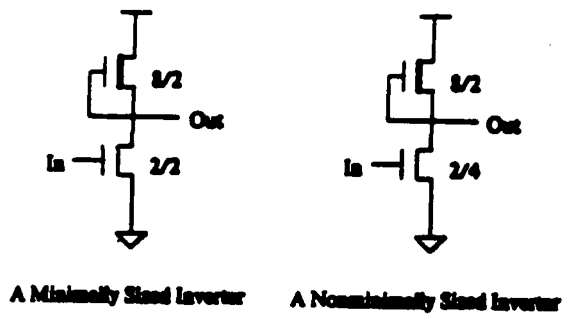


Figure 2.7: Two Differently Sized Inverters. The minimum-sized inverter does not work for all inputs that the nonminimum inverter works for. It will only work if its input has not suffered a threshold drop. The nonminimum inverter works for a greater range of inputs at the cost of using more area and being slower than the minimum sized inverter.

---

threshold drop device in the transistor models. Other systems, *e.g.*, RSIM, have not modelled threshold drops.

Races A race occurs between two signals when the final state of circuit is a function of which signal changes first. Often the proper operation of a circuit requires that when a race happens, one of the two signals always changes first. Races can be modelled because the signal model explicitly includes time. Some systems, such as MOSSYM, do not model time, and therefore can't detect races.

Hazards Hazards can be considered as degenerate cases of races where a signal is racing with itself. A hazard occurs when a circuit can detect the number of transitions a signal has during settling. For example, if a signal begins and ends at zero, but the circuit it's in has a different final state depending on whether the signal has zero or two transitions, then a hazard occurs when the signal goes through two transitions.

2.5.2 Bugs Not Modelled

Bugs which can't be modelled by Silica Pithecus are those arising from unanticipated capacitive coupling and spikes on off-chip inputs.

Ignoring Capacitive Coupling

Capacitive coupling is ignored so that number and size of a node's possible nets is minimized. If capacitive coupling were modeled, then nets would span the chip

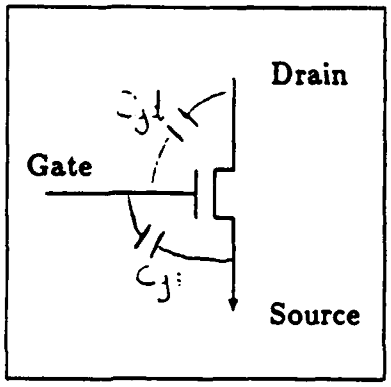


Figure 2.8: MOS transistor symbol with explicit C_{gs} and C_{gd}.

and the circuit would have to be analyzed monolithically.[6] This analysis would be infeasible for even moderately sized circuits. In this section I show the sources of capacitive coupling, discuss the drawbacks in ignoring it.

The two main sources of capacitive coupling are the gate to source capacitance of MOS transistors, and the coupling of physically close wires. The dominant source is the capacitance of transistors, by far.

A MOS n-channel transistor is modelled in detail as having a capacitor between its gate and source (Figure 2.8) and between its gate and drain. This capacitance can be increased by the inclusion of an explicit capacitor. This capacitor forms a coupling between the gate and source nodes of the transistor through which current can flow.

There are two undesirable effects of ignoring coupling capacitance: not being able to model anticipated capacitive coupling, and not being able to catch bugs arising from unanticipated capacitive coupling. The first undesirable effect, not being able to model circuits which depend on capacitive coupling can be overcome by making such circuits primitives of the design system. They can then be treated specially, as other primitives (*e.g.*, transistors) are. For example, bootstrap drivers could be treated this way. No other large scale analysis system (*e.g.*, TV [TV, Jouppi], MOSSIM [Bryant81], RSIM [Terman], Crystal [Ousterhout]) has handled capacitive coupling in a general way. For example, [Terman] does a prescan of the circuit which recognizes bootstrap drivers and replaces them with primitive

[6]There is no way within my framework for heuristically examining only those parts of the circuit where capacitive coupling effects are significant. There are two reasons for this. First, I am trying to be formal so that I can make clear statements about what it means to verify a circuit. The inclusion of heuristics interferes with this goal. Second, there is no clean mechanism for a designer to indicate parts of the circuit which bear closer inspection by the verifier.

elements. Other than *ad hoc* checks, there does not seem to anything that can be done about the inability to catch bugs arising from unanticipated capacitive coupling.

No Loads on Off-chip Inputs

Charge retention analysis does not model wavering off-chip inputs; there are no edges in a net transition graph corresponding to off-chip inputs which suddenly undergo large loads. Therefore off-chip inputs must be considered immune to on-chip transitions. For example, Gnd and VDD are considered constant; even if a heavy load is placed on them this load cannot be detected anywhere in the circuit. The same is true for the other off-chip inputs. I do not know if this is a significant problem for MOS circuits (*i.e.*, if circuits have failed because designers didn't anticipate loads causing off-chip inputs to waver), but I know of no system which addresses this problem.

2.6 Summary

This chapter presented the the component model, signal model, and network model of Silica Pithecus. These models are known collectively as a circuit model. The bugs which can and can't be modelled within this model were discussed.

This chapter introduced the notion of net behavior. The idea of focusing analysis on nodes and the nets they are members of during settling is novel. Previous researchers have looked at the nets a circuit is partitioned into during settling without investigating what the partitioning looks like from a node's point of view. The power of this idea in the next chapter appears in the next chapter.

Chapter 3

Multilevel Verification

Circuit behavior can be described at different levels of abstraction ranging from the analog level to the functional level. Multilevel verification proves that two different levels of behavioral description describe the same behavior. The more abstract level description of a design's behavior is called its *abstract behavior*, the less abstract level description of a design's behavior is called its *concrete behavior*.

Multilevel verification has two parts: *abstraction* and *validation*. Abstraction lifts the concrete behavior level to the abstract behavior level. The resulting description is called the *abstracted behavior*. Validation is the comparison of the abstracted and abstract behaviors. An implementation *meets* its specification if the comparison succeeds, otherwise the implementation is declared *incorrect*.

This two step scheme, abstraction followed by validation, makes the dependence of verification on the abstraction function explicit. The abstraction function serves two purposes: approximation and structure definition. Which purpose is served depends on the descriptive level of the concrete behavior. At the lower levels, such as the signal level, abstraction replaces "physical reality" with gross approximations (*i.e.*, digital values). At this level, the accuracy of verification is bounded by the accuracy of the abstraction function. For higher descriptive levels, such as the digital level, abstraction replaces data structures with the values they denote. For example, when abstracting from the boolean level to the arithmetic level, a vector of booleans is abstracted to a number.

Not all elements of the concrete domain map into elements of the abstract domain. The abstraction function is responsible for returning an error value on elements which do not map. For example, the abstraction of a signal that can not be abstracted to either 0 or 1 must be an error value. One role of constraints is to specify that a design's inputs and outputs are always abstractable. For example, at the signal level, the constraints such as no-drop, `falls-first`, `overpowers`, and `control` all exist solely to ensure that input and output signals abstract to non-error values.

The difficulty of validation depends on the descriptive level. Digital validation consists of proving two digital expressions are equivalent. Except for complex combinational circuits such as ripple carry adders, the formulas are small and easily compared. Functional validation, which must compare formulas in domains richer than boolean algebra (*e.g.*, arithmetic), is hard to do.

A design is ***digitally verified*** when its signal behavior has been verified against its intended digital behavior. Likewise, a design is *functionally verified* when its digital behavior has been verified against its intended functional behavior. We are primarily concerned with digital verification. The methods to be proposed for digital verification have been implemented, but no such claim is made for proposed methods of functional verification.

A major advantage of multilevel verification is that the description of the intended behavior itself can be verified. Abstract behavior at one level is concrete behavior at the next abstraction level up and can itself be verified. This is a powerful mechanism for verifying the behavior of a design over many levels of description.

This chapter has four sections. In the first section some of the levels of behavioral description are discussed. A brief discussion of the different descriptive levels of time is included in the first section. The second section describes multilevel verification. It delineates issues and presents my theory of multilevel verification. The third section focuses on digital verification and functional verification. The chapter concludes with a summary.

A given circuit can be described in a variety of ways. For example, a circuit can be described at the signal, digital, gate, or functional level. Each level hides details of the lower level.[1] A pictorial example appears in Figure 1.1. The higher levels of description are more concise than the lower levels.

We are concerned with the signal, digital, and, to some degree, functional descriptions of circuits. The signal level describes a circuit's function in terms its input and output signals. The digital level is similar to the boolean level, but with the inclusion of the operand *float* and the function *join*. The functional level is defined by the user.

The signal level views a circuit as a map from input signals to output signals. The signals at state nodes are both input signals and output signals. A signal is what appears on your oscilloscope when you stick the probe on a node. It is crucial that signals, rather than voltages at some point in time, are employed as the primitive values operated on by circuits. Signals encompass time, therefore any method for abstracting signals into digital values must consider time.[2] If just voltage at some point in time were used, then a separate mechanism would be required to handle time. A separate mechanism would make the formalization of verification much

[1]The different descriptive levels do not form a linear hierarchy. For example, a digital design that uses tristate buses cannot be abstracted to the gate level.

[2]Even if an abstraction function ignores time, it must do so explicitly.

harder. The following chapters discuss in detail the derivation of a circuit's signal behavior from its structure.

The digital level has three primitive values: 0, 1, (called *boolean* values) and *float*. It has the standard boolean functions for operating on boolean values and the operator join. Join is an n-ary function which returns *float* if all its arguments are *float*, returns 1 if all the non-*float* values are 1, returns 0 if all the non-*float* values are 0, and otherwise signals an error. Join and *float* are used to model circuits containing tristate devices.[3]

The functional descriptive level is very abstract. Its primitive types and operations are defined by the user. For example, the primitives may be numbers, pointers, or sets. The corresponding operations might be addition, dereferencing, or union. This level is characterized by being very close to the application the designer is creating a circuit for. Specifying the behavior of a circuit at this level is very concise and simple.

3.1 A Formal Model of Verification

We first introduce notation. *Design* is the object whose behavior is to be verified. Its behavior in some domain D is written $B_D(Design)$. *Design*'s inputs are are written $\hat{\imath}_D = i_{1_D}, \ldots, i_{n_D}$ where $\hat{\imath}_D \in D^n$. Its outputs are the result of applying its behavior to its inputs: $\hat{o}_D = B_D(Design)(\hat{\imath}_D) = o_{1_D}, \ldots, o_{m_D}$ where $\hat{o}_D \in D^M$. The two domains of a given multilevel verification are called *con* and *abs*. Multilevel verification relates $B_{con}(Design)$ and $B_{abs}(Design)$. We will often leave the domain descriptor off the inputs and outputs and just write $\hat{\imath}$ and $\hat{o}$. When domain descriptors are elided the *con* domain is assumed.

SPEC, the specification of *Design*, is what it to be proved about *Design*.[4] P are the set of promises about how the design will be used, they restrict the class of allowable inputs to *Design*. P corresponds to what we have been calling constraints. *Design* is verified with respect to its specification *SPEC* and constraints P when the *verification condition*

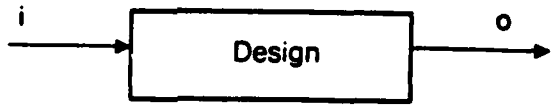
$$\forall_{\hat{\imath} \in D^n}\{(\bigwedge_{p \in P} p(\hat{\imath})) \Rightarrow SPEC(\hat{\imath}, B_D(Design)(\hat{\imath}))\}$$

is shown to hold.

[3]The hardware digital level is the abstraction level espoused in *Introduction to VLSI Systems* [Mead/Conway]. This book provides a methodology of design (*i.e.*, approved ways of connecting parts together) which preserves the digital abstraction. For example, a designer who learned only from their book would not create a bootstrap driver (Figure 12.1) or a three transistor ram (Figure 8.2).

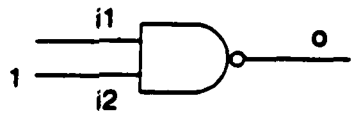
[4]*SPEC* has a very specific form when multilevel verification is performed, as will be seen below.

- *Design* is a circuit with one input i and one output o ranging over time-varying signals S. $B_S(Design) : S \rightarrow S$



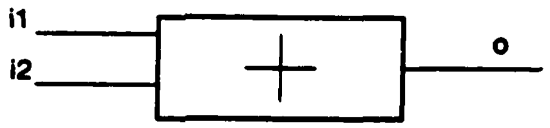
$$\forall_{i \in S}\{stable(i) \Rightarrow stable(o)\}$$

- *Design* is a nand gate.



$$\forall_{i_1, i_2 \in \{true, false\}}\{i_2 \Rightarrow (o = not(i_1))\}$$

- *Design* is an adder whose inputs $\hat{\imath}$ and output o range over the natural numbers N. In this example P is empty.



$$\forall_{i_1, i_2 \in N}\{o = i_1 + i_2\}$$

Figure 3.1: Example Verification Conditions. These are all examples of monolevel verification.

Examples of verification conditions are shown in Figure 3.1. The first example checks properties about signals in to and out of a circuit. It specifies that if the input signal is stable then so is the output signal. The next example is the statement that if you hold one of the inputs to a nand gate high then the gate looks like an inverter to the other input. The third example is the statement that some device is an adder.

All of these are examples of what we call *monolevel verification*. In monolevel verification only one level of behavioral description is handled. Monolevel verification attempts to prove specific properties of the behavior of a device.

Multilevel Verification

In multilevel verification there are two different levels of description for *Design*, $B_{con}(Design)$ and $B_{abs}(Design)$, which must be related. The relationship between the two domains is given the by abstraction function ABS. It maps elements of *con* into elements of *abs*. There are some elements of *con* for which ABS is undefined. We make ABS a total function on *con* by adjoining a special "error element" $\perp$, to ABS's range as a target for ABS when it would otherwise be undefined. Specifically, $ABS : con \Rightarrow abs + \{\perp\}$. ABS is *defined* for some input i when $ABS(i) \in abs$. Any element of *con* for which ABS returns $\perp$ is said to be *invalid*.

When discussing specific domains, ABS may be subscripted with its types. For example, when the context isn't clear, the abstraction function from signals S to tristate values T will be written $ABS_{S\rightarrow T}$. Finally, we also have $ABS(\hat{\imath}) = ABS(i_1), \cdots, ABS(i_n)$

Given $B_{con}(Design)$ and ABS, $B_{abs}(Design)$ is projected from $B_{con}(Design)$ by the following formula:

$$\forall_{\hat{\imath}\in con^n}\{\mathrm{ABS}(B_{con}(Design)(\hat{\imath})) = B_{abs}(Design)(\mathrm{ABS}(\hat{\imath}))\}$$

wherever ABS($\hat{\imath}$) and ABS($B_{con}(Design)(\hat{\imath})$) are both defined.

This relationship can be expressed as a commutative diagram (Figure 3.2). The key point is that $B_{abs}(Design)$ does not exist independently of ABS. For example, if ABS returns $\perp$ everywhere, then $B_{abs}(Design)$ is nowhere defined.

We can now define multilevel verification. Given $B_{con}(Design)$ and of *Design*'s intended abstract behavior $IB_{abs}(Design)$, we wish to show $IB_{abs}(Design) = B_{abs}(Design)$:

Definition: Multilevel Verification is Verification where the specification is of the form

$$\mathrm{ABS}(B_{con}(Design)(\hat{\imath}_{con})) = IB_{abs}(Design)(\mathrm{ABS}(\hat{\imath}_{con})).$$

where $IB_{abs}(Design)$ is the intended *abs* level behavior of *Design* and where both sides of the equation are defined.

$$\begin{tikzcd}
\hat{i}_{abs} \arrow[r, "B_{abs}(Design)"] & \hat{o}_{abs} \\
\hat{i}_{con} \arrow[u, "ABS"] \arrow[r, "B_{con}(Design)"'] & \hat{o}_{con} \arrow[u, "ABS"']
\end{tikzcd}$$

Figure 3.2: The relationship between $B_{con}(Design)$ and $B_{abs}(Design)$. Given ABS and $B_{con}(Design)$, $B_{abs}(Design)$ is projected from $B_{con}(Design)$ according to this diagram. The commutativity need only hold when ABS($\hat{\imath}_{con}$) and ABS($B_{con}(Design)(\hat{\imath}_{con})$) are defined.

Putting everything together yields

$$\forall_{\hat{\imath}\in con^n}\{\bigwedge_{p\in P} p(\hat{\imath}) \Rightarrow \mathrm{ABS}(B_{con}(Design)(\hat{\imath})) = IB_{abs}(Design)(\mathrm{ABS}(\hat{\imath}))\}$$

as the verification condition for multilevel verification.

The Role of P, and Automatically Generating Constraints

An important part of the above proposition are the promises P, also called constraints. P restricts the class of allowable inputs to a design. The constraints have three roles:

1. To ensure valid inputs (*i.e.*, that ABS($\hat{\imath}_{con}$) is defined). These are called *input constraints* and are assumed to be met.

2. To ensure valid outputs (*i.e.*, that ABS($B_{abs}(Design)(\hat{\imath}_{con})$) is defined). These are called *output constraints* and can be automatically generated.

3. To specialize behavior. These are called *logical constraints* and must be provided by the designer.

Input constraints are assumed to be met by the environment in which a design operates. Inputs are assumed to be valid because they either come from the outside world, in which case we must assume they are valid, or they are outputs from other verified designs, which guarantees they are valid. For example, consider a circuit. Only inputs from off-chip must be assumed to be valid, all other inputs are valid because they are outputs of verified subcircuits.

The constraints which ensure outputs are valid can be automatically generated. Given a representation of ABS($B_{abs}(Design)(\hat{\imath}_{con})$) that makes error conditions explicit, constraints are generated which ensure the conditions causing the error conditions never arise. For example, in the signal domain, this how threshold constraints are generated. When an input signal suffering a threshold drop causes an invalid output signal the signal will be constrained from suffering a threshold drop. The constraint guarantees the output will not be invalid (at least from this particular cause). An explicit example is presented below.

The constraints which specialize the behavior of a design must be provided by the designer. They cannot be assumed or automatically generated. For example, the logical constraints a designer must provide to Silica Pithecus to declare certain inputs are mutually exclusive. The constraints specializing behavior must be employed when deriving a design's behavior from its structure. Otherwise, behaviors which don't arise will be predicted.

The Big Picture

Three major ideas lurk in the verification condition for multilevel verification. They are:

- The relationship between different behavioral descriptions.
- The relationship between structure, function, and context.
- Intentional analysis.

The first item, the relationship between two different behavioral descriptions has been discussed in detail.

The introduction of this dissertation made the claim that structure alone does not determine function, but that structure and context do. Referring to the verification condition of multilevel verification, we can now state more precisely the relationship be structure, function, and context: P is the context that the structure *Design* must have to exhibit behavior $IB_{abs}(Design)$.

Intentional analysis says that what must be derived from a design's structure is not its behavior, but the abstraction of that behavior. That is, the important item is not $B_{con}(Design)$, but ABS($B_{con}(Design)(\hat{\imath}_{con})$). As a result, the full behavior of *Design* need not be generated. For example, the focus of Silica Pithecus's behavior generation is not to generate signal behavior, but to generate constraints which ensure that whatever the signal behavior is, it will be abstractable. Much of the signal behavior is left symbolic (*i.e.*, in terms of the names of signals rather than the signal values themselves).

We now look at functional and digital verification in detail. Functional verification will be presented first to get the reader accustomed to the ideas and notations.

3.2 Functional Verification

The application of the design (*e.g.*, numbers, sets, pointers) determines the type of abstraction and functional validation. A given design may support many domains at once. There are two major problems associated with functional verification: validation and accurate specification. In the following we will concentrate on just the arithmetic domain (integers and addition) to illustrate functional verification.

Validation, the comparison of abstracted functional behavior and intended functional behavior, is difficult to do. A significant amount of knowledge of the functional domain must be employed.[5] Embedding in a verifier the knowledge needed is hard because it is hard to *a priori* determine which knowledge will be useful; embedding everything is virtually impossible.

The second problem with functional verification is accurately specifying behaviors. A specification must be well defined before validation can be done. For example, consider the digital specification of a 16 bit adder for adding unsigned integers, called A and B. Each integer is represented as vectors of bits (the standard representation). There are several choices possible for handling overflow. The overflow bit can be ignored, it can be appended to the sum to yield a 17 bit result, or it can be returned separately to signal an error condition.

Even though each of the three methods of handling the overflow bits still results in an adder, only one, the second, can be specified as $A + B$ (for $A, B < 2^{16}$). The others, which are far more common, must be specified differently, for example, $(A+B) \bmod 2^{16}$ for the first method. (There's even residual vagueness in the specifications given. The function $+$ is polymorphic, its behavior depends on the types of its operands. A more accurate specification is $A +_{int} B$ and $(A +_{int} B) \bmod 2^{16}$.)

Example: An Adder Cell

Consider a full adder (Figure 3.3). It has three inputs A, B, and *Carryin*, and two outputs called *Carryout* and *Sum*. The Adder Cell is composed of gates, therefore its concrete behavior is expressed in the boolean domain. We with to show the adder cell adds numbers, therefore its abstract behavior is expressed in

[5]Also, these domains are very complex and formulas in them may be undecidable.

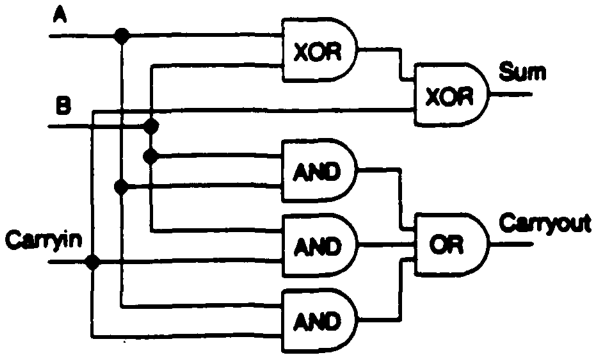


Figure 3.3: A Full Adder

the arithmetic domain. The two domains are called $B*$, for boolean values (*true* and *false*) plus vectors of booleans, and W, for the whole numbers:

$D_{con} = \{\mathtt{true}, \mathtt{false}, [b_0, \ldots, b_n]\}$
$D_{abs} = W(\textit{Whole Numbers}).$

Our abstraction function is

$ABS(true) = 1$
$ABS(false) = 0$
$ABS([b_0, \ldots, b_n]) = \sum_{0 \leq i \leq n} 2^i ABS(b_i)$

The Adder Cell's concrete behavior is

$$B_{B*}(\text{Adder Cell}) = \lambda \text{ a b c . [sum(a, b, c), carry(a, b, c)]}.$$

The Adder Cell only accepts Booleans, it does not accept vectors. This fact is expressed by the input promise

$$Boolean(i_1, i_2, i_3).$$

The Adder Cell's behavior in the arithmetic domain is intended to be

$$IB_W(AdderCell) = \lambda \text{ a b c . a + b + c}.$$

The Adder Cell's specification is therefore

$$ABS((\lambda \text{ a b c . [sum(a, b, c), carry(a, b, c)]})(i_1, i_2, i_3))$$

$$= (\lambda \text{ a b c . a + b + c})(ABS(i_1), ABS(i_2), ABS(i_3))$$

and the verification condition for verifying the Adder Cell is

$$\forall_{i \in B}.\{Boolean(i_1, i_2, i_3) \Rightarrow$$

$$ABS((\lambda \text{ a b c . } [\text{sum(a, b, c), carry(a, b, c)}])(i_1, i_2, i_3))$$

$$= (\lambda \text{ a b c . a + b + c})(ABS(i_1), ABS(i_2), ABS(i_3))\}$$

Using beta-reduction, the above equation can be simplified to

$$\forall_{i \in B}.\{Boolean(i_1, i_2, i_3) \Rightarrow$$

$$ABS(sum(i_1, i_2, i_3)) + 2ABS(carry(i_1, i_2, i_3))$$

$$= ABS(i_1) + ABS(i_2) + ABS(i_3)\}$$

This equation can be shown to hold by exhaustively enumerating all combinations of the inputs, thereby verifying the design of the adder cell.

An example of hierarchical verification in the arithmetic domain appears in Chapter 10. That example verifies a multi-bit adder built from Adder Cells.

3.3 Digital Verification

Digital verification verifies that the signal behavior of a circuit correctly implements the intended digital behavior of the circuit. S, the signal domain, and T, the tristate domain, are the two levels of behavioral description of digital verification. As mentioned previously, signals are characterized as mapping time into a voltage and a strength, $S : time \rightarrow (voltage \times strength)$. The tristate domain is {`false`, `true`, `float`}. We have $ABS_{S \rightarrow T} : S \rightarrow T$. In this section we assume some magic function which yields $B_S(Design)$ for any *Design*. Later chapters discuss how $B_S(Design)$ is generated.

This section uses as examples only combinational circuits (*i.e.*, combinations of gates) with electrically isolated inputs. This simplifies the abstraction function and allows us to concentrate on the process and major ideas of digital verification. Later chapters handle circuits with both state and non-electrically isolated inputs. Signal strength is also ignored here as it is irrelevant for the verification of combinational circuits.

The abstraction function we will use in the example is

$$\begin{aligned}
\text{ABS} &= \lambda\, s\,.\, \text{VTB}(voltage(s(t_f))) \\
\text{VTB} &= \lambda\, v\,.\, v > 3.5 \rightarrow \texttt{true}, \\
&\qquad v < 1.5 \rightarrow \texttt{false}, \\
&\qquad \bot.
\end{aligned}$$

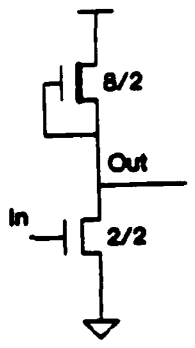


Figure 3.4: A Minimum Sized Inverter. To ensure that the output of this device is valid we must guarantee the input does not suffer a threshold drop. This device has one input signal and one output signal.

ABS accepts a signal S and turns its final voltage into a boolean value using the rule that voltages above 3.5 volts give `true`, voltages less than 1.5 volts give `false`, and other voltages give $\perp$.

Silica Pithecus is given the schematic *Design*, a specification of *Design*'s intended tristate behavior $IB_T(Design)$, and the logical constraints on it Design. We wish to show $IB_T(Design) = B_T(Design)$. The verification condiction for digital verification is

$$\forall_{\imath \in S}\{\bigwedge_{p \in P} p(\imath) \Rightarrow \mathrm{ABS}(B_S(Design)(\imath)) = IB_T(Design)(\mathrm{ABS}(\imath))\}$$

where P = Input Constraints $\cup$ Output Constraints $\cup$ Logical Constraints.

The input constraints, which guarantee that inputs are valid, are assumed. The output constraints, which guarantee outputs are valid, are automatically generated. Only one type of constraint, the threshold constraint, is needed for verifying combinational circuits with isolated inputs. The logical constraints are given.

Example: A Minimally Sized Inverter

Consider a minimally sized inverter (Figure 3.4). We wish to prove that this device computes the boolean `NOT` function. The inverter takes one input signal and yields one output signal.

The procedure which generates $B_S(Design)$ from *Design* uses the knowledge that ABS only looks at the final value of a signal. The procedure yields

B_S(Inverter) =
  λ *in* .

$$\begin{array}{l}
\lambda\, t\, . \\
\quad t = t_f \rightarrow \\
\qquad (\text{voltage}(in(t)) = 5 \rightarrow 1, \\
\qquad\ 3.5 \leq \text{voltage}(in(t)) < 5 \rightarrow 1.66, \\
\qquad\ 1.5 \leq \text{voltage}(in(t)) < 3.5 \rightarrow X, \\
\qquad\ 0 \leq \text{voltage}(in(t)) < 1.5 \rightarrow 5), \\
\qquad \mathbf{X}
\end{array}$$

as the signal behavior of the inverter. This is read as follows. The Inverter takes a signal *in* and returns a signal, call it *out*. For any time not equal to t_f, *out* returns X, meaning an unknown voltage. When time is t_f, *out* returns a voltage based upon the circuit model (*cf.* Chapter 2) and the input voltage. No more detail than this is necessary to verify the inverter.

Inverter's intended boolean behavior is

$$IB_T(Inverter) = \lambda\, a\, .\, \texttt{NOT}(a).$$

Finally, we have the constraint that Input is valid. The constraint guarantees that Input's final voltage is either above 3.5 volts or below 1.5 volts. The constraint is written as

$$VV(\text{voltage}(in(t_f)))$$

where VV stands for Valid Voltage.

All these formulas are wrapped together to create the verification condition for Inverter:

$$\begin{array}{l}
\forall_{i \in S}\{VV(\text{voltage}(i(t_f))) \Rightarrow \\
\\
\text{ABS}((\lambda\, in\, . \\
\qquad \lambda\, t\, . \\
\qquad\quad t = f \rightarrow \\
\qquad\qquad (\text{voltage}(in(t)) = 5 \rightarrow 1, \\
\qquad\qquad\ 3.5 \leq \text{voltage}(in(t)) < 5 \rightarrow 1.66, \\
\qquad\qquad\ 1.5 \leq \text{voltage}(in(t)) < 3.5 \rightarrow X, \\
\qquad\qquad\ 0 \leq \text{voltage}(in(t)) < 1.5 \rightarrow 5), \\
\qquad\qquad \mathbf{X} \\
\quad (i)) \\
\\
= (\lambda\, a\, .\, \texttt{NOT}(a))(\text{ABS}(i))\}
\end{array}$$

This equation can be simplified by using beta-substitution and distribution of functions across the arms of the conditional. The result is

$$\begin{array}{l}
\forall_{i \in S}\{VV(\text{voltage}(i(t_f))) \Rightarrow \\
\text{voltage}(i(t_f)) = 5 \rightarrow \texttt{false},
\end{array}$$

$3.5 \leq \text{voltage}(i(t_f)) < 5 \rightarrow \perp,$
$1.5 \leq \text{voltage}(i(t_f)) < 3.5 \rightarrow \perp,$
$0 \leq \text{voltage}(i(t_f)) < 1.5 \rightarrow$ **true**
= NOT(ABS(i)) }

The input constraint is used to eliminate the third clause.[6] It guarantees that the predicate of the third clause will always be false. Throwing out the third clause yields

$\forall_{i \in S}\{VV(\text{voltage}(i(t_f))) \Rightarrow$
$\text{voltage}(i(t_f)) = 5 \rightarrow$ **false**,
$3.5 \leq \text{voltage}(i(t_f)) < 4.5 \rightarrow \perp,$
$0 \leq \text{voltage}(i(t_f)) < 1.5 \rightarrow$ **true**
= NOT(ABS(i))}.

The second clause shows an error condition which arises when the final value of the input signal is between 3.5 and 5 volts. This condition is indicative of a threshold drop. The error condition is prevented by generating the constraint that *input* cannot suffer a threshold drop. The constraint is written no-drop(voltage($i(t_f)$)). This constraint subsumes VV(voltage($i(t_f)$)). Posting this constraint yields

$\forall_{i \in S}\{$no-drop(voltage($i(t_f)$)) $\Rightarrow$
$\text{voltage}(i(t_f)) = 5 \rightarrow$ **false**,
$0 \leq \text{voltage}(i(t_f)) < 1.5 \rightarrow$ **true**
= NOT(ABS(i)) }

Because voltage($i(t_f)$) = 5 implies $ABS(i)$ and it is the only clause to do so, we replace the former with the latter. Similarly $0 < \text{voltage}(i(t_f)) < 1.5$ implies NOT(ABS(i)) and it, too, is so replaced. These substitutions yield

$\forall_{i \in S}\{$no-drop(voltage($i(t_f)$)) $\Rightarrow$
ABS(i) $\rightarrow$ **false**,
NOT(ABS(i)) $\rightarrow$ **true**
= NOT(ABS(i)) }

Finally, the upper conditional is recognized as an idiom for NOT and rewritten to

$\forall_{i \in S}\{$no-drop(voltage($i(t_f)$)) $\Rightarrow$
NOT(ABS(i))
= NOT(ABS(i)) }

which is true.

[6] In the implementation the constraint is used to prevent the third clause from even being generated. It easier to explain how to get rid of something than to explain why it's not created in the first place.

3.4 Summary

There are many levels of description for the behavior of circuits. Multilevel verification is the successful comparison of the same behavior at two different levels of description. The abstraction function was the key to relating the two behavioral descriptions.

The verification condition for multilevel verification made explicit many informal notions presented in the introduction of this dissertation. These include the relationship between different behavioral descriptions, the relationship between structure, function, and context, intentional analysis, the role of constraints, and a method for automatically generating constraints.

An idealized scenario of how Silica Pithecus verifies a combinational circuit was presented. This scenario stressed the abstraction of signals and the automatic generation of constraints.

Chapter 4

State and Time

Circuits with state are more complex than combinational circuits. Unlike combinational circuits whose outputs can be predicted solely from the final values of their input signals, the outputs of circuits with state depends on their input signals' entire waveforms. This chapter motivates and explains the abstraction function used by Silica Pithecus which enables Silica Pithecus to verify circuits with state. This chapter also develop the input constraints that ensure that all output signals are abstractable.

There are two properties we desire of the abstraction function. First, we want the abstraction function to be *deterministic*. That is, it should guarantee that the same inputs (after abstraction) yield the same outputs (again, after abstraction).

Second, we want to be able to symbolically predict, for all valid states and inputs of a circuit, the final state of the circuit. The abstraction function found will therefore require that nodes storing state are not corrupted.[1] State and corruption are technical terms. State is stored charge that affects the outputs of a circuit. Corruption is not so easily defined, although it does correspond to the usual intuitive idea. (Corruption is defined below.) There are pathological circuits that are designed to corrupt state. Such circuits cannot be verified by Silica Pithecus. Fortunately, such circuits are rare.

Just as the abstraction function given in the previous chapter required threshold constraints to ensure outputs were valid, the abstraction function we are looking for will require *timing constraints* to ensure outputs are valid. There are two types of timing constraints, `stable-after` constraints and `falls-first` constraints. A `stable-after` constraint requires a given signal to be stable after some other signal is asserted. A `falls-first` constraint requires a given signal to fall before some other signal changes. These are the only constraints needed to ensure stored charge

[1]The general, and correct, statement is that nets which store state are not corrupted. A node is the degenerate case of a net with only one node. We will soon talk in terms of both nets and nodes storing charge.

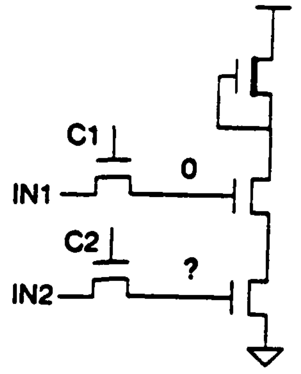


Figure 4.1: A NAND Gate. When the upper transistor "stores" a 0 the charged stored on the lower transistor can be safely corrupted (*i.e.*, C2, can glitch).

is not corrupted. A control signal is a signal that must remain asserted once it is asserted. That is, control(N_s) is shorthand for stable-after(N_s, N_s).

Silica Pithecus treats state bits as both inputs and outputs of a circuit. The abstraction procedure is applied not only to the output signals of a circuit, but also to its state signals as well.[2] The signals of nodes which do not store state, do not gate transistors, and which are not outputs of circuits are not abstracted. Such nodes, called *connection nodes*, serve only as vias for information flow. Because connection nodes are never abstracted, the charge on them can be, and often is, freely corrupted.

This chapter has nine sections. The first section discusses locating a circuit's state nodes. The second section defines corruption. The third section develops the idea of deterministic abstraction functions and shows why the abstraction function of the previous chapter is nondeterministic. The fourth section gives the abstraction function used by Silica Pithecus. The fifth section presents the representation which makes invalid signals apparent. It also shows how constraints ensuring signals are valid are issued. The following sections expand on the fifth section in different ways. The sixth section discusses some implementation issues. The seventh section presents a fairly complicated example. The eight section generalizes the methods to handle circuits containing transistors with different (positive) thresholds. The chapter ends with a summary.

4.1 Finding State in a Circuit

A node storing charge is a state node if its stored charge can affect a circuit's output behavior. State nodes appear in memory cells, shift registers, and latches.

[2] A state signal is the signal at a node storing state.

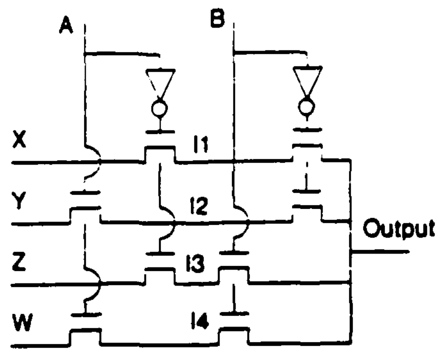


Figure 4.2: A 4-way Selector. The charge stored on the internal nodes is often corrupted during settling. The internal nodes are called *connection nodes.*

It is difficult to accurately assess which stored charge represents state. Data dependencies allow what would otherwise appear to be state to be freely corrupted. For example, consider an nMOS NAND gate with latched inputs whose upper transistor is "storing" a 0 (Figure 4.1). In this case the stored charge on the lower gate can be corrupted (by $C2_s$ glitching) without affecting the outputs of the NAND gate. Silica Pithecus does not consider such data dependencies when locating state. If there is any point in time at which a node stores state, then Silica Pithecus considers the node to store state at all times. This approach is overly conservative, but no problems have yet been encountered using it.

State nodes are statically discovered by performing a recursive tree scan of the net behavior equations of a circuit. The scan starts from the output nodes of the circuit. For each node N in the scan, N's possible nets are examined. If a net is not driven then all its *dominant nodes* are state nodes. The dominant nodes of a net are those whose capacitance overpowers[3] the capacitance of the other nodes in the net. If a net contains an input node then the input node is the only dominant node of the net. Except for nets containing inputs nodes, a net can contain an arbitrary number of dominant nodes. The scan is repeated for each new dominant node found and for each node mentioned in the predicates of N. The scan ends when no new state nodes are found.

Nodes which don't store state and whose signals don't appear in the predicates of net behavior expressions (*i.e.*, don't gate transistors) are connection nodes. Connection nodes can be freely corrupted. For example, consider a 4-way selector (Figure 4.2). Assume that the four inputs are driven and that both $A_b(t_i)$ and $B_b(t_i)$ are not asserted. If, during a computation, A_b and B_b waver before returning to being unasserted, then the internal nodes will be corrupted. This corruption has no effect on the output of the selector.

[3]Silica Pithecus defines overpowers as ten times greater.

4.2 Corruption

Let *net* be some final net.[4] We want to be able to predict the voltage of any node in *net* solely from the initial state of the nodes in *net*. If *net* contains driven nodes then the prediction can be based solely the driven nodes. If *net* contains no driven nodes, then the final voltages can be precisely predicted from initial voltages if and only if, during settling, no nodes in *net* shared charge with nodes not in *net*. When a final net contains nodes which have shared charge with nodes not in the final net, we say that the net has been *absolutely corrupted*. A later section discusses avoiding corruption.

Example: Latched Inverter Cell

For example, consider again the Latched Inverter Cell (Figure 2.2). Its S node has three possible nets, [S], [S, Out, Vdd], and [S, Out, Vdd, Gnd], all of which can be final.[5] When $S_{net}(t_f)$ is either [S, Out, Vdd] or [S, Out, Vdd, Gnd] Thevenin equivalents are used to calculate the voltage at S. When $S_{net}(t_f) = [S]$, then absolute corruption may have occurred. Consider the computation sequences of Figure 4.3. In the upper figure $S_{net}(t_f)$ ($= [S]$) is not corrupted because S has not shared charge with any other nodes. (Although it began connected to other nodes, it did not share charge with them. Remember, it is assumed that circuits start out at steady state.) In the lower figure [S] has been corrupted because it momentarily shares charges with Gnd and VDD. Another path (not shown) which corrupts [S] is [S, Vdd, Out] $\rightarrow$ [S, Vdd, Out, Gnd] $\rightarrow$ [S]. In this case S shares charge with Gnd before being isolated.

Absolute corruption is too strict a notion. Instead, a less strict form, simply called *corruption*, is more useful. A final net is *corrupted* if its dominant nodes have shared charge with nodes not in the net. Corruption differs from absolute corruption in that the charge sharing of non-dominant nodes is not considered. The voltages of nodes in a net whose non-dominant nodes have uncertain charge can be calculated with sufficient accuracy for verification.

Dominant nodes cannot be allowed to share charge with any extra nodes, regardless of their capacitance. They might lose enough charge a little at a time such that an accurate prediction of voltages on final nets is impossible.

[4]Review the section on signals in Chapter 2 for the definition of final net.

[5]Not all of a net's possible nets can be final. Steady state logical constraints dictate that although certain nets can occur during settling, these nets can't occur at steady state.

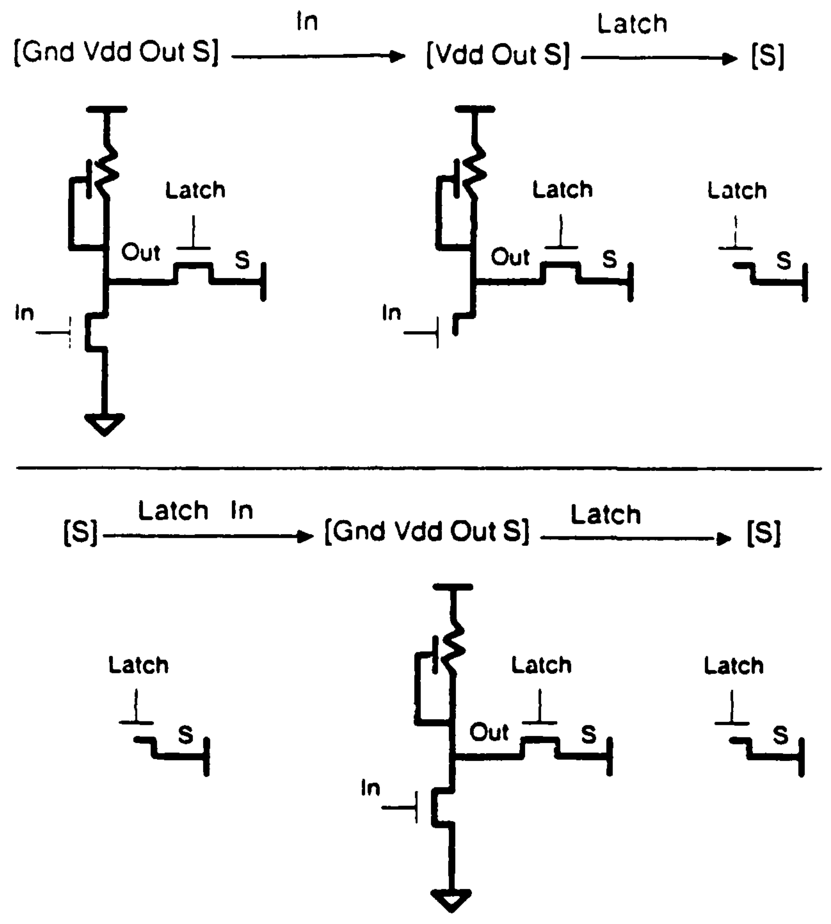


Figure 4.3: Computation Paths. These are two possible computation paths of the Latched Inverter Cell's S node. In the upper path S begins connected to Vdd and Out. Then the input to the inverter goes low and then Latch falls isolating S. In the lower path S begins isolated. Then Latch goes high (while In is asserted) then falls, leaving S isolated. The labels of the edges indicate which signal causes the transition and the signal's direction of change.

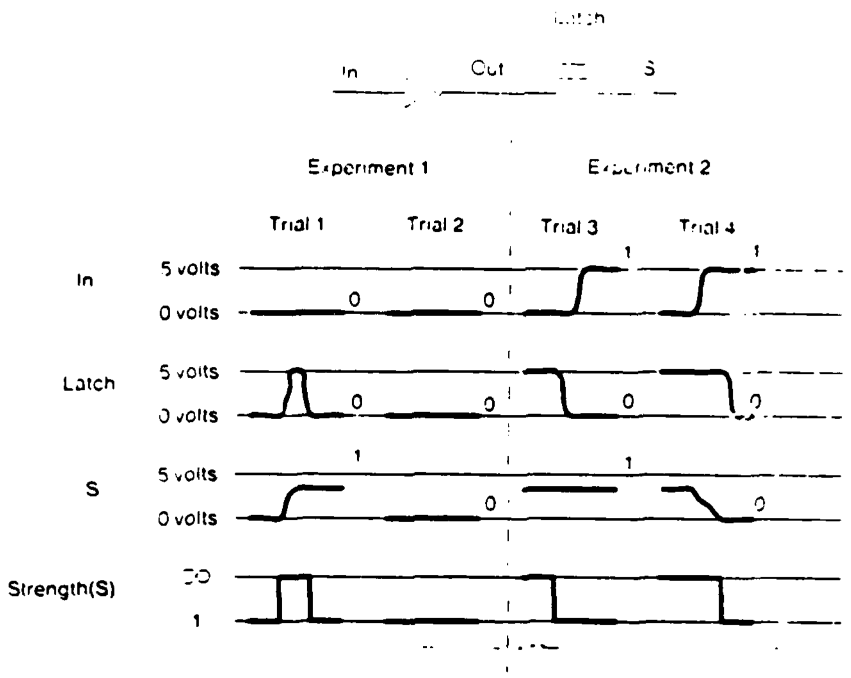


Figure 4.4: Non-deterministic Abstraction. This figure depicts two different experiments (trials 1 and 2 trials 3 and 4). Each experiment shows equivalent inputs giving different outputs. "Non-deterministic behavior" refers to this phenomenon.

4.3 Deterministic and Nondeterministic Abstraction Functions

From a theoretical viewpoint, the major problem of the abstraction function of the previous chapter, $\lambda\ s$. VTB(voltage($s(t_f)$)), is that it projects a *non-deterministic* B_T. A behavior is non-deterministic when the same inputs can give different outputs. Two idealized experiments with the Inverting Latch (Figure 2.2) will make this clear. Consider Figure 4.4, which shows four different sets of inputs applied to the Latched Inverter Cell, and shows the S_s which results from the inputs. Both experiments show the boolean behavior of this device giving different outputs for the same inputs. In the first experiment ABS(In_s) and ABS($Latch_s$) are both 0, but ABS(S_s) is either 1 or 0. In the second pair of experiments, ABS(In_s) is 1 and ABS($Latch_s$) is 0, but again ABS(S_s) is either 1 or 0.

There are two ways to project a deterministic B_T. The first is to have the abstraction function differentiate between inputs which cause different outputs, *i.e.*, guarantee the inputs will be different when the outputs are different. The second, and more natural method, is to declare certain output signals to be invalid. Silica Pithecus takes this approach. Its abstraction function will declare S_s in trials 1 and

4 to be invalid. As a result, when verifying the circuit, Silica Pithecus constrains Latch$_s$ from glitching and constrains In$_s$ from changing before Latch$_s$ falls. These constraints prevent the invalid signals from occurring.

Circuit designers will recognize the first pair of runs as exhibiting what is known as a "hazard" and the second pair of runs as exhibiting a "race." Hazards occur when control signals go through too many transitions during settling. The standard example (exhibited in the figure) of a hazard is a control signal which begins at 0 and ends at 0, but goes through two transitions (up and down).[6] Because the transitions of a control signal affect a circuit as much as the value of a control signal, the extra transitions cause the circuit to end up in the wrong state.

Races occur in MOS technologies when the signal isolating an undriven net and the signal changing the value of the net both change during the same computation. The order in which the signals change affects the final state of the net. If the net is first isolated, its final value will be different than if it is changed before being isolated. For example, Latch$_s$ of the Latched Inverter Cell must fall before In$_s$ changes in order to isolate [S] before it is changed by In$_s$. When In$_s$ and Latch$_s$ can change on the same computation there is a *race condition*.

Hazards and races cause corruption. If hazards are outlawed, and races carefully ordered[7] then corruption will not occur and a deterministic B_T results. The ordering of races is presented below. The signals which Silica Pithecus declares to be invalid are precisely those which represent (potentially) corrupted state.

4.4 The Abstraction Function

The abstraction function has the desirable properties that it projects a deterministic B_T and it places timing constraints on inputs to ensure outputs are valid.

4.4.1 Valid Signals

The rules for valid signals are presented below. If all output signals (state signals are considered output signals) are valid then no state nodes are corrupted. A signal is valid when

- Its strength does not decrease after its voltage changes.[8]

[6] I often think control signals are called such not because they control things, but because they themselves must be controlled.

[7] Dictating which of two signals in a race must change first *orders* the race. Timing constraints order races.

[8] This is not the least restrictive rule for ensuring freedom from corruption. There are a range of rules, from simple to complex, that can be used for avoiding corruption. Simpler rules are easier to check than complex rules, but they give more false negatives than complex conditions

- Its final voltage is less than 1.5 volts or more than 3.5 volts.

The two rules are called the *strength rule* and *voltage rule*, respectively.

Sources of corruption can divided into two classes: *Momentary Expansion Corruption*, and *Driven Corruption*. Momentary Expansion Corruption occurs when a net expands and then contracts. Except when the net contains driven nodes both before expanding and after contracting, the expansion and contraction shows up as an increase in strength followed by a decrease in strength. An example of this type of corruption is shown in the lower half of Figure 4.3. We conservatively assume that the voltage of signal always changes when its strength increases.[9] The Strength Rule prevents momentary expansion corruption.

The `stable-after` constraint is used to prevent momentary expansion corruption. It prevents a net from shrinking after it has expanded. The form of the constraint is `stable-after`(X_s,Y_s), which means that after X_s is asserted, Y_s must remain stable.[10] It also implies that Y_s must be stable as X_s is asserted (*i.e.*, as X_s rises, in nMOS). `control`(x_s) is shorthand for `stable-after`(x_s,x_s).

Driven Corruption occurs when the initial strength of a signal is ∞ and the voltage of the signal changes before the strength of the signal becomes finite. For example, considering the Latched Inverter's S node again, the sequence [S, Vdd, Out, Gnd], [S, Vdd, Out], [S], is an example of Driven Corruption. Driven Corruption is prevented by the Strength Rule which dictates that once a signal's voltage changes the signal's strength cannot decrease.

The `falls-first` constraint is used to prevent Driven Corruption. It ensures that strength falls before voltage changes. `falls-first`(X_s,Y_s) means that if X_s must fall before Y_s changes.

The two constraints, `falls-first` and `stable-after`, are collectively called *timing constraints*. The two constraints collectively outlaw hazards and order races. Races are ordered such that undriven nets are isolated before they are changed. We call the ordering of races prescribed by the Strength Rule the *natural ordering*.

do. This rule was chosen as the simplest rule which verifies most.circuits. Interested readers can invent more complex rules on their own. Silica Pithecus is as accurate as possible with doing a data-dependent analysis.

[9]Heuristic data dependent analysis could be performed to allow Silica Pithecus to be less conservative. This is discussed in more detail in Chapter 6.

[10]Bouncing can be allowed by treating the assertion of X_s as taking some finite amount of time rather than the assertion being instantaneous. If we take "after X_s is asserted" to mean "after X_s is finished being asserted" and require Y_s to be stable while X_s is being asserted, then bouncing can be fit into the model.

4.5 Issuing Constraints

The representation of signal behavior makes error conditions (*i.e.*, invalid signals) explicit so that constraints guaranteeing valid signals can be automatically generated. The representation is based on ***computation paths*** and ***net transition graphs***. Constraints are generated directly from each state node's net transition graph.

Unlike threshold constraints which always apply, timing constraints conditionally apply. That is, only under certain conditions must X_s fall before Y_s or must X_s be stable after Y_s is asserted. Associated with each timing constraint is a predicate on the state of the circuit which controls the application of the constraint. Only when the predicate is true must the constraint hold. For example, in the scenario of the first chapter had the constraint $\text{Latch}_b \Rightarrow$ overpowers($[\text{In}]_{In}$,$[\text{S}]_S$). The predicate is "Latch_b" and the constraint itself is "overpowers($[\text{In}]_{In}$,$[\text{S}]_S$)."

The elements of the predicates are boolean signals, rather than boolean values, and are meaningless unless unless their time component is quantified. We conservatively state that a predicate is true if there is any moment time when it is true. That is, for some predicate P, $P \equiv \exists t[P(t)]$.

This section has two parts. It first discusses computation paths and net transition graphs. It then gives the algorithms for generating timing constraints.

4.5.1 Computation Paths and Net Transition Graphs

Computation paths and ***net transition graphs*** are formal tools used by steady state behavior validation. A computation path is the list of nets, in order, that a node is a member of during a computation. Figure 4.3 shows two different computation paths for the S node of the Latched Inverter Cell (Figure 2.2). In the upper example S begins connected to Vdd. Then Latch_s falls isolating S. In the lower example S begins isolated. Then Latch_s goes high (while Input_s is high) and then low, isolating S again. The labels of the edges indicate which signal causes the transition and its direction of change. For transistors that turn on, the label also includes a predicate describing the net that merges with the net undergoing the change. For example, the first edge of the lower path indicates not only that Latch_s is asserted, but that it does so when In_s is asserted. Had In_s not been asserted, the resulting net would not have included Gnd.

The possible computation paths of a given node are formalized as paths through a net transition graph. A node undergoes a ***net transition*** when the net it is a member of changes due to some transistor changing state. For example, referring again to Figure 4.3, the change from net [S] to net [Gnd Vdd Out S] is a net transition. The vertices of a net transition graph for node N correspond to the possible nets of N, and the edges correspond to net transitions of N.[11] The vertices

[11]To avoid ambiguity a net consists of *nodes* and *conducting transistors* whereas a graph consists of

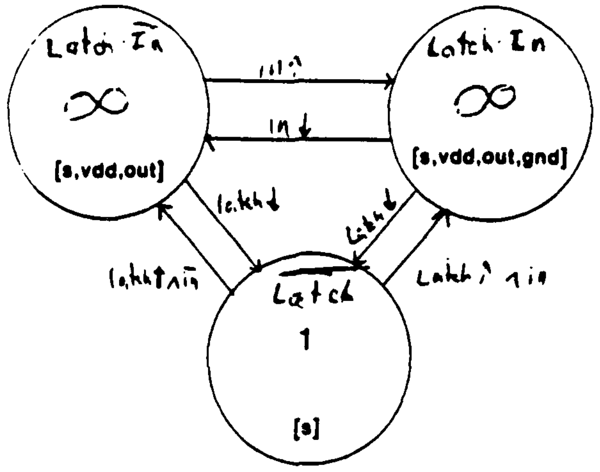


Figure 4.5: The Net Transition Graph of the S node of Latched Inverter Cell.

are labelled with the predicate selecting the net corresponding to the vertex. The edges are labeled as they are labelled in computation paths. A computation path is a traversal through a net transition graph.

Each vertex in a net transition graph is annotated with the strength of the signal it corresponds to. A transition in the net transition graph from a vertex to a less strong vertex is called a *downward transition*, to a higher strength vertex is called an *upward transition*, and to an equal strength vertex a *sideways transition*. Finally, two vertices *A* and *B* are *neighbors* if there is some path of sideways transitions connecting them.

For example, consider again the Latch Inverter Cell (Figure 2.2). The net behavior equation of its S node is

$$\begin{aligned} S_{net} = \lambda\ t\ .\ & \text{Latch}_b(t) \wedge \text{In}_b(t) \rightarrow [\text{S Vdd Out Gnd}], \\ & \text{Latch}_b(t) \wedge \texttt{NOT}(\text{In}_b(t)) \rightarrow [\text{S Vdd Out}], \\ & \texttt{NOT}(\text{Latch}_b(t)) \rightarrow [\text{S}] \end{aligned}$$

The net transition graph (Figure 4.5) for the S node of this cell has three vertices corresponding to S's three possible nets. Because each possible net can be transformed by a single transistor state change into one of the other possible nets, there are six edges in the graph. There are two downward edges for when Latch falls, and two corresponding upward edges, for when Latch rises. The are two sideways edges corresponding to In's two states.

vertices and *edges* (also called *arcs*).

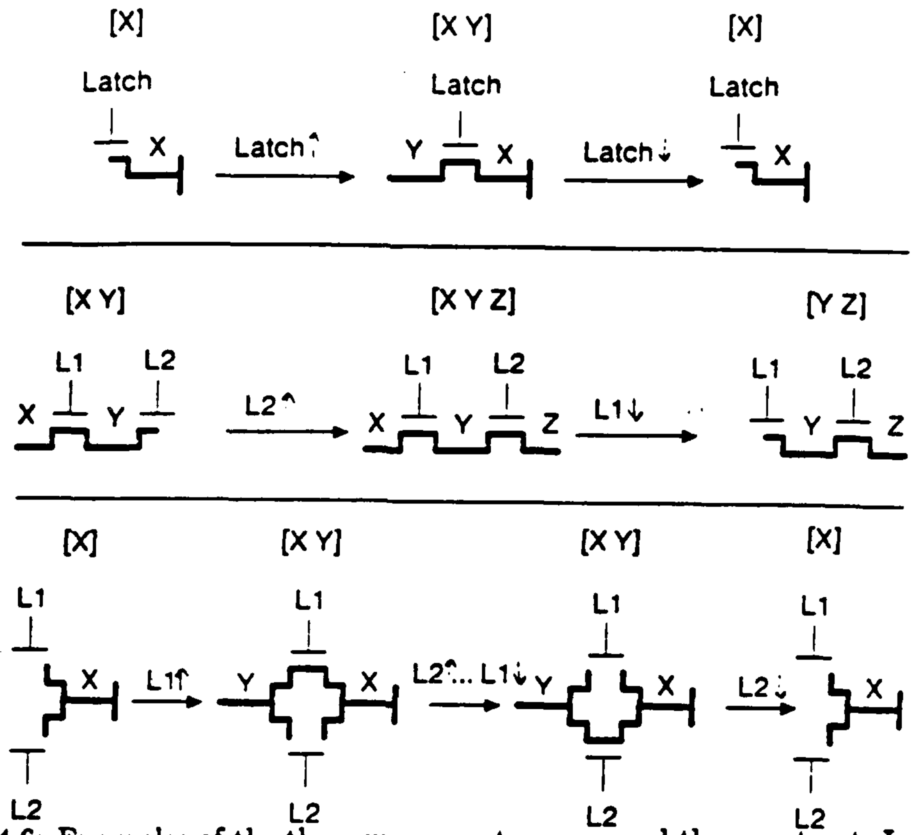


Figure 4.6: Examples of the three ways a net can expand then contract. In the first diagram an upward transition is followed by a downward transition where both transitions are caused by the same transistor. In the second diagram an upward transition is followed by a downward transition where the two transitions are due to different transistors. In the third diagram an upward transition is followed by two sideways transitions, then a downward transition is made.

4.5.2 Ensuring Strength Never Decreases After Increasing

stable-after constraints ensure a signal's strength never decreases after increasing. When a signal's strength decreases after increasing the signal has undergone Momentary Expansion Corruption.[12] Momentary expansion corruption is avoided by forbidding downward net transitions to follow upward net transitions.

The ways a signal's strength increases and then decreases can be divided into three classes (Figure 4.6). The first occurs when a control signal (defined below) is asserted then unasserted (this is called a *hazard*). The second cause of momentary expansion corruption occurs when an upward transition is followed by a downward transition where the upward and downward transitions are caused by different sig-

[12]We are punning between corruption of nets and corruption of signals. We say N, is corrupted if and only if N is a dominant node of a corrupted net.

nals. The third cause of momentary expansion corruption occurs when an upward transition is followed by some number of sideways transitions then followed by a downward transition.

All sources of momentary expansion corruption are avoided by the following requirement.

> **Rule 1:** All signals causing downward transitions from a vertex V cannot change after any signal causing an upward transition into V or into a neighbor of V is asserted.

`stable-after` constraints, which ensure this rule is always met, are generated by the following algorithm. The algorithm is explained using the notation "$P|A$" which means "The predicate P simplified under the assumption that A is asserted." For example, assuming that b is asserted by making it true, `(and a b c)`|b reduces to `(and a c)`, and b|b reduces to true.

Algorithm Generate `stable-after` Constraints:

Inputs: The net transition graph G of node N.

Outputs: The `stable-after` constraints which ensure N_s does not undergo Momentary Expansion Corruption.

Method:

```
For each vertex $V$ in $G$ do
  Let $P$ be the predicate choosing $V$.
  For each downward outgoing edge $D$ from $V$ do
    Let $SD_s$ be the signal causing $D$.
    For each upward edge $U$ into $V$ or into a neighbor of $V$ do
      Let $SU_s$ be the signal causing $U$.
      Issue the timing constraint
      "$P|SU_b \Rightarrow$ stable-after($SU_e$, $SD_e$)." ■
```

For example, consider again the Latched Inverter Cell (Figure 2.2), and the net transition graph of its S node (Figure 4.5). The above algorithm generates the following two `stable-after` constraints on Latch:

- $In_b \rightarrow$ `stable-after`($Latch_e$, $Latch_s$)
- $\neg In_b \rightarrow$ `stable-after`($Latch_e$, $Latch_s$).

These two constraints are collapsed into `stable-after`($Latch_e$, $Latch_s$), which is then simplified to `control`($Latch_s$). A control signal is a signal that, once asserted, must remain asserted.

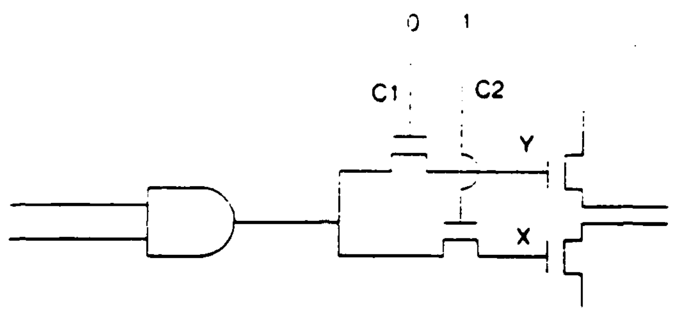


Figure 4.7: Voltage Changes. Under pessimistic enough conditions, asserting C1 can pull X low.

4.5.3 Ensuring Strength Never Decreases After the Voltage Changes

As mentioned above, Driven Corruption occurs when the initial strength of a signal is ∞ and the voltage of the signal changes before the strength of the signal becomes finite. For example, considering the Latched Inverter (Figure 2.2) again, S_s undergoes Driven Corruption when the computation path [S, Vdd, Out, Gnd] $\rightarrow$ [S, Vdd, Out] $\rightarrow$ [S] occurs. In this computation path S_b starts out at 0 but goes to 1 before being isolated.

Driven Corruption is avoided by requiring a signal's strength to decrease before the signal's voltage changes. Unfortunately, a completely strict enforcement of this will make it impossible to verify many reasonable circuits. A strict interpretation of this rule will disallow driven nets to expand before contracting. For example, consider a gate routed to two latches (Figure 4.7). Assume that $C1_b(t_i)$ is unasserted and $C2_b(t_i)$ is asserted. Also assume that $C1_b$ and $C2_b$ can change during the same phase. If $C2_b$ becomes unasserted before $C1_b$ is asserted, no problems arise. If $C1_b$ goes high first, however, then X_s may suffer a large voltage change depending on the capacitance of Y and the driving force of the gate.

Circuits such as this are common. We therefore ignore the effects of purely capacitive nodes and declare that voltage changes do not occur between *comparable driven nets*. Two nets are comparable if and only if they be transposed into each other by adding and subtracting undriven nodes from their periphery (Figure 4.8 shows an example). We now state the rule that is used to prevent Driven Corruption.

> **Rule 2:** Signals causing downward net transitions from a driven vertex V must change before signals which cause sideways transitions to incomparable nets change.

The following algorithm enforces Rule 2 by issuing `falls-first` constraints.

Algorithm Generate `falls-first` Constraints:

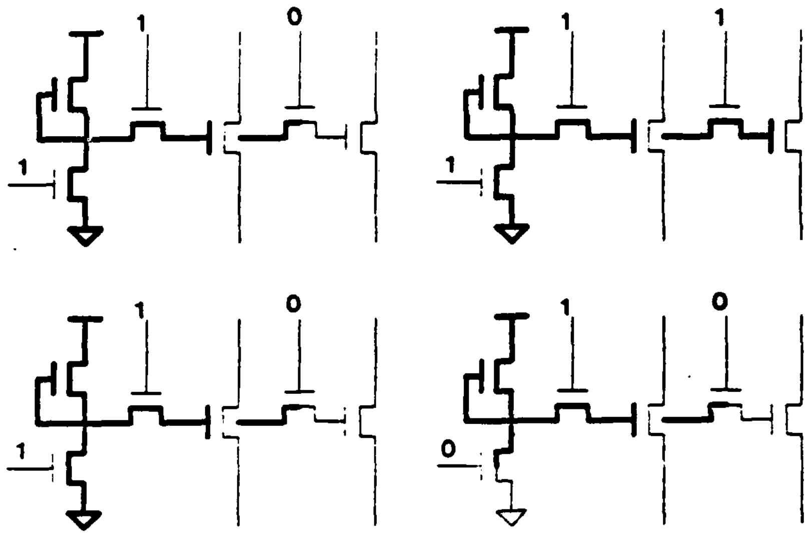


Figure 4.8: Comparable and Incomparable Nets. Two driven nets are comparable if they can be transposed into each other by adding and subtracting undriven nodes at their periphery. The two nets in the first half of the figure are comparable. The two nets in the bottom half are incomparable.

Inputs: The net transition graph G of node N.
Outputs: The `falls-first` constraints which prevent Driven Corruption of N_s.
Method:

```
For each vertex $V$ in $G$ do
  Let $P$ be the predicate choosing $V$.
  If $V$ corresponds to a driven net then
    For each downward outgoing edge $D$ from $V$ do
      Let $SD_s$ be the signal causing $D$.
      For each sideways edge $U$ of out of $V$ into an incomparable vertex
        Let $SU_s$ be the signal causing $U$.
        Issue the constraint "$P \wedge \neg SD_b(t_f) \Rightarrow$ falls-first($SD_s$,$SU_s$)." ■
```

For example, consider again the Latched Inverter Cell's S node and its net transition graph (Figure 4.5). The above algorithm generates the following two `falls-first` constraints.

- $\mathrm{In}_b \wedge \mathrm{Latch}_b \wedge \neg \mathrm{Latch}_b(t_f) \Rightarrow$ **falls-first**(Latch_s, In_s)
- $\neg \mathrm{In}_b \wedge \mathrm{Latch}_b \wedge \neg \mathrm{Latch}_b(t_f) \Rightarrow$ **falls-first**(Latch_s, In_s)

These two constraints are merged to yield

$$\mathrm{Latch}_b \wedge \neg \mathrm{Latch}_b(t_f) \Rightarrow \texttt{falls-first}(\mathrm{Latch}_s, \mathrm{In}_s).$$

The predicate is abbreviated `falls`(Latch_b) to yield the final form of the constraint: `falls`(Latch_b) $\Rightarrow$ `falls-first`(Latch_s, In_s). This constraint prevents the voltage on S from changing before S is isolated.

Figure 4.9 restates the restrictions and merges the two algorithms together.

4.6 Implementation Issues

There are two notes on the implementation that bear mentioning. First, Silica Pithecus does not perform a timing analysis. It therefore cannot determine the relative ordering of changes in signals. Second, the designer envisioned data dependencies between control signals are usually much coarser than the data dependencies embedded in constraints. This observation is exploited by trying to prove timing constraints hold under coarser data dependencies than under which they are predicated. This results in faster, simpler proofs.

Silica Pithecus does not perform a timing analysis; it cannot prove that some signal changes before some other signal. Therefore Silica Pithecus cannot verify circuits relying on races. Silica Pithecus proves that Y_s falls before X_s changes (*i.e.*, satisfies `falls-first` constraints) by showing that X_s is stable when Y_s can fall. Likewise, Silica Pithecus proves that X_s doesn't change after Y_s rises (*i.e.*, satisfies `stable-after` constraints) by showing that X_s is stable when Y_s can rise. The lack of a timing system leads to false negatives. These false negatives can be avoided by incorporating a timing system into Silica Pithecus. However, since circuits which rely on races are rare, this is not a major drawback.

The dependencies intended by the designer are simpler than the data dependencies in timing constraints. A designer often intends a restriction on a signal to hold under all conditions, but the timing constraints only restrict the signal under a few specialized conditions. For example, where timing constraints require a signal to be a control signal only at certain times, a designer often intends the signal to always be a control signal. A second example is a signal which constraint generation requires to be a control signal during computation C, but which the designer intends to be stable during C.

Silica Pithecus exploits the simpler data dependencies of real designs by attempting to satisfy timing constraints using simpler (more restrictive) data dependencies than they are predicated under. This is advantageous because it is easier to satisfy timing constraints using simpler data dependencies. For example, when Silica Pithecus has to prove that Y_s is a control signal on P, it first tries to show that Y_s either is stable on P or that Y_s is always a control signal. Only if these attempts fail does it try to prove Y_s is a control signal on P.

When proving some restriction is true on P Silica Pithecus will sometimes try to prove the restriction holds even when P is false. For example, often the only

Rule 1: All signals causing downward transitions from a vertex V cannot change after any signal which causes an upward transition into V or into a neighbor of V is asserted.

Rule 2: Signals causing downward net transitions from a driven vertex V must change before signals which cause sideways transitions to incomparable nets change.

Algorithm GENERATE TIMING CONSTRAINTS:

Inputs: The net transition graph G of node N.
Outputs: The constraints which ensure a signals strength does the right thing.
Method:

```
For each vertex $V$ in $G$ do
  Let $P$ be the predicate choosing $V$.
  For each downward outgoing edge $D$ from $V$ do
    Let $SD_s$ be the signal causing $D$.
    [Generate stable-after constraints.]
    For each upward edge $U$ into $V$ or into a neighbor of $V$ do
      Let $SU_s$ be the signal causing $U$.
      Issue the constraint
      "$P|SU_b \Rightarrow$ stable-after($SU_s$, $SD_s$)."
    [Generate falls-first constraints.]
    If $V$ corresponds to a driven net then
      For each sideways edge $U$ of out of $V$ into an incomparable vertex
        Let $SU_s$ be the signal causing $U$.
        Issue the constraint
        "$P \wedge \neg SD_b(t_f) \Rightarrow$ falls-first($SD_s$, $SU_s$)." ■
```

Figure 4.9: The Restrictions on Net Transitions and Their Implementation.

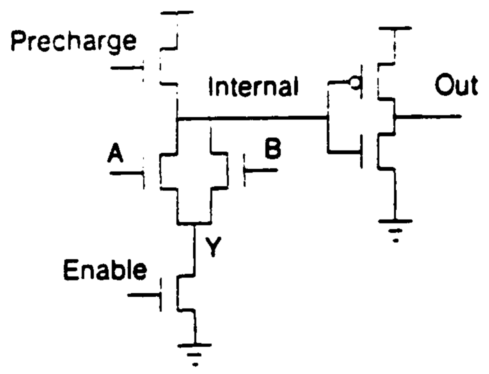


Figure 4.10: An OR Gate Implemented in CMOS Domino Logic. Node Internal is precharged on ϕ_1. Enable goes high on ϕ_2. A_s and B_s must be stable when Enable goes high.

relevant term of P is that some clock is asserted. If the clock is asserted, then the restriction will hold regardless of whether P itself is true.

4.7 A Complex Example

As a more complex example consider an OR gate designed with CMOS Domino Logic [Krambeck] (Figure 4.10). It is used as follows. On one phase Precharge_s is held high and enable_s is held low. During this time A_s and B_s are free to settle. On the next phase Precharge_s is low while Enable_s is high. During this time A_s and B_s are stable. The relationship between signals informally stated above are automatically discovered, and stated as timing constraints.

Consider now the net transition graph of the Internal node of this device (Figure 4.11).[13] This graph has five vertices and twelve transitions. The timing constraints generated for this graph appear in Table 4.1. The numbers in this figure correspond to the numbered vertices in the OR gate's net transition graph.

The timing constraints are summarized and compared against the designer's restrictions in Table 4.2. The only restrictions that match completely are the ones requiring Precharge_s to be a control signal. The other timing constraints depend on more factors than the designers restrictions. The designer intends A_s and B_s to be stable when Precharge_b is low, they only change when Precharge_b is asserted. The timing constraints only require A_s and B_s to be control signals when Precharge_b

[13]The graph in the figure is not the real graph, which has 11 vertices and approximately 40 arcs. The graph shown was generated from the real graph by merging vertices whose associated nets had the same nodes. This causes multiply connected nodes to be represented by only one vertex rather than three.

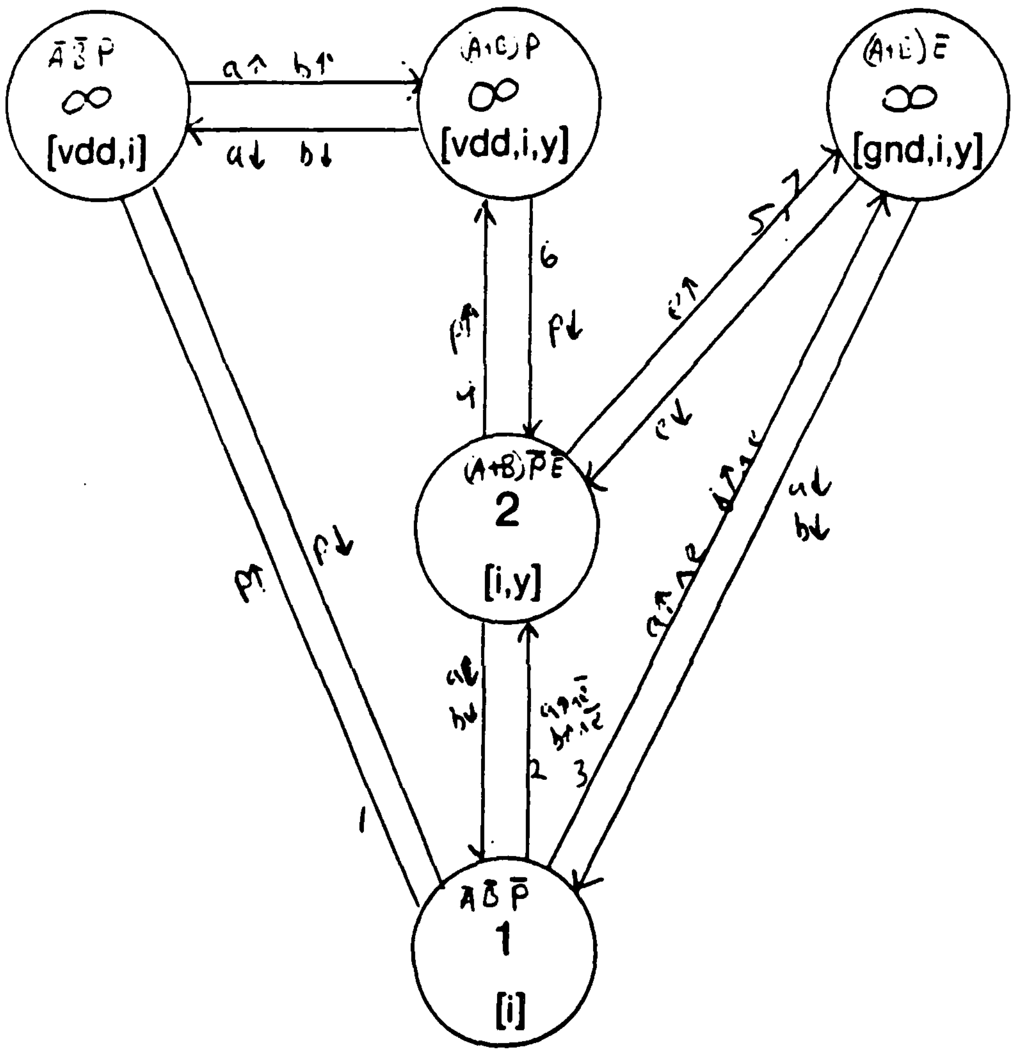


Figure 4.11: The Net Transition Graph of Internal Node of the CMOS Domino Logic Or Gate. This is an abbreviated form of the real net transition graph. In this graph vertices with same set of nodes are merged.

1. $\mathtt{NOT}(A_b) \wedge \mathtt{NOT}(B_b) \rightarrow \mathtt{control}(\mathrm{Precharge}_s)$

2. $A_b \vee B_b \rightarrow \mathtt{control}(\mathrm{Precharge}_s)$

3. $A_b \vee B_b \rightarrow \mathtt{control}(\mathrm{Enable}_s)$
   $\mathrm{Enable}_b \rightarrow \mathtt{control}(A_s)$ and $\mathtt{control}(B_s)$
   $A_b \rightarrow \mathtt{stable\text{-}after}(\mathrm{Enable}_s, A_s)$
   $B_b \rightarrow \mathtt{stable\text{-}after}(\mathrm{Enable}_s, B_s)$
   $\mathrm{Enable}_b \rightarrow \mathtt{stable\text{-}after}(A_s, \mathrm{Enable}_s)$
   $\mathrm{Enable}_b \rightarrow \mathtt{stable\text{-}after}(B_s, \mathrm{Enable}_s)$

4. $\mathtt{NOT}(A_b) \wedge \mathtt{NOT}(\mathrm{Precharge}_b) \rightarrow \mathtt{control}(B_s)$
   $\mathtt{NOT}(B_b) \wedge \mathtt{NOT}(\mathrm{Precharge}_b) \rightarrow \mathtt{control}(A_s)$

Table 4.1: Timing Constraints for the Domino Logic OR Gate.

is low, rather than to be stable. Also, where the designer intends Enable_s to be a control signal, the timing constraints only force it to be a control signal when A_b or B_b is high.

4.8 Generalizing to Multiple Types of Transistors

This section generalizes the ideas to include circuits containing multiple types of transistors. The theory readily extends to transistors which conduct when their gates are "off" (at 0 volts) such as p-channel transistors. (Silica Pithecus only handles nMOS circuits with only one type of positive threshold transistor.)

The theory does not readily handle circuits containing transistors with different positive thresholds. In particular, it doesn't correctly handle *multiply connected nodes* where the *redundant* transistors have different thresholds. Nodes X and Y are multiply connected when there is more than one path of conducting transistors between them. The transistors which make the multiple paths possible are called redundant.

For example, consider a cMOS *transmission gate* (Figure 4.12). This device is used as a threshold drop free switch [Weste]. When A_b is asserted both transistors are conducting. If X is at Vdd, Y will eventually be pushed to Vdd across the p-channel transistor. If X is at Gnd, Y will eventually be pulled down to Gnd across the n-channel transistor. If both transistors are conducting (which occurs when A_s changes value) there are two connections between X and Y. If either transistor stops conducting, X and Y will still be connected because the other transistor is

Generated Constraints	Designer Restrictions
control(Precharge$_s$).	Same

Enable$_b$ $\rightarrow$ control(A$_s$)
A$_b$ $\wedge$ B$_b$ $\wedge\neg$Precharge$_b$ $\rightarrow$ control(A$_s$)
stable-after(Enable$_s$,A$_s$)
stable-after(A$_s$,Enable$_s$)
	$\neg$Precharge$_b$ $\rightarrow$ stable(A$_s$)

Enable$_b$ $\rightarrow$ control(B$_s$) A$_b$ $\wedge$ B$_b$ $\wedge\neg$Precharge$_b$ $\rightarrow$ control(B$_s$) stable-after(Enable$_s$,B$_s$) stable-after(Enable$_s$,B$_s$)	$\neg$Precharge$_b$ $\rightarrow$ stable(B$_s$)
A$_b$ $\vee$ B$_b$ $\rightarrow$ control(Enable$_s$)	control(Enable$_s$)

Table 4.2: Comparing the Designer Restrictions and the Generated Restrictions.

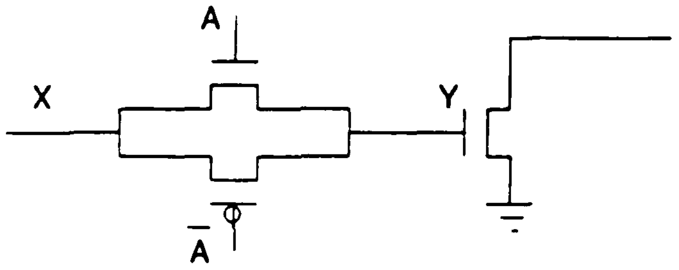


Figure 4.12: cMOS Transmission Gate. This is an example of a circuit which gives rise to multiply connected nodes. This circuit occurs frequently in practice.

still on. Therefore, there are three configurations of the transistor states yielding a net containing X and Y.

As an example of circuit where multiply connected nodes cause problems, consider the circuit of Figure 4.13. In this figure there are two types of positive threshold transistors. The transistor gated by B is of one type and the other transistors are of the other type. The transistor gated by B has a threshold (called Thresh-B) which is lower than threshold of the transistor gated by A (called Thresh-A). Assume that all transistors are initially off and that N is at ground. If the circuit undergoes the transitions A high, B high, B low, my method will calculate the wrong voltage for N, namely VDD − Thresh-A. The correct voltage is VDD − Thresh-B. This bug arises because the original rules did not account for different threshold drops.

The correction is to treat some sideways transitions as if they were upwards transitions. This is done by assigning to vertices secondary numbers based on the types of transistors in them. These secondary numbers can be used to order vertices

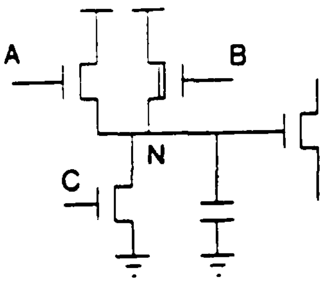


Figure 4.13: A Circuit Containing Two Types of Positive Threshold Transistors. All the transistors are n-channel. The threshold of the transistor gated by A is **thresh-A** and the threshold of the transistor gated by B is **thresh-B**. **Thresh-B** is smaller than **thresh-A**.

just as strength is used to order vertices. Transistors with smaller thresholds would have higher secondary numbers associated with them. With this subnumbering scheme the original rules of computation paths can still be used. Vertices would first be ordered by strength and secondarily by types of transistors.

4.9 Summary

Signals are abstracted into digital values by first making sure they are valid, and then turning their final values into digital values. A signal N_s is valid when $N_{net}(t_f)$ is not corrupted.

An explicit representation for signal behavior, called net transitions graphs was developed. This representation makes invalid signals explicit. It also yielded a fast and effective method for automatically generating constraints which guarantee all output signals will be valid.

The method was extended to handle all MOS processes. The extension only required a more precise ordering of the possible nets of a circuit.

Chapter 5

Non-isolated Inputs

This chapter confronts non-electrically isolated inputs. The Inverting Latch of Chapter 1 is an example of a circuit with a non-electrically isolated input. When Latch is asserted, the input node In is connected to node S. This connection affects the input signal applied to In. There are two ramifications of non-electrically isolated inputs. The first is ensuring information flows from an input node into a circuit, rather than vice versa. The second is augmenting the generation of timing constraints to accommodate nets which contain input nodes.

Another type of constraint, called a *flow constraint*, is needed to ensure that input nodes act as inputs. An input node N of circuit C acts as an input only when the signal S applied to N is not *materially changed* by C. S is materially changed by connection to a C when the abstraction of S when it is connected to C and the abstraction of S when it is not connected to C differ. When S is materially changed by C then N is acting as an output node. Input signals are usually materially changed by momentary connections to driven nodes and by charge sharing bugs.

This chapter has two sections. The first section discusses flow constraints. The second discusses issuing timing constraints for nets which contain input nodes.

5.1 Flow Constraints

Flow constraints dictate that signals must flow into input nodes and out of output nodes. A node acts as input of circuit C only when signals flow into C circuit through it. Likewise, a node acts as an output of circuit C only when signals flow out of C through it. The generation of flow constraints requires access to a circuit's intended digital behavior. The intended digital behavior defines which nodes are inputs and which are outputs.

For example, consider again the Inverting Latch of Chapter 1. Assume that the capacitance of its S node is 100pf. Further assume that its Input node is connected to a precharged bus whose capacitance is only 50pf. When the Latch goes high, signal flow will be from S to Bus, thereby changing Bus. This change is contrary to the designer's intentions of Bus changing S. In this scenario, Input is acting as an output node, not as an input node.

Signal flow is constrained by the *overpowers* constraint. There are two ways a signal value X can overpower a signal value Y: when X's strength is sufficiently greater than Y's strength,[1] or when X and Y have the same logical value. The first condition can be checked statically without recourse to the "run-time" behavior of a circuit. A signal X overpowers a signal Y when $\forall t$ [overpowers(X$_s$(t), Y$_s$(t))].

Flow constraints can be formally motivated by changing the abstraction function. The new abstraction function rejects input signals which were materially changed by the circuit to which they were input. We will not develop the formal aspects of motivating flow constraints. The interested reader is encouraged to conduct this investigation on her own.

The following notation is used in this chapter. Let $Net = [N_1, \ldots, N_n]$ be a net. Net_{N_i} represents the signal value computed by Net at node N_i.

5.1.1 Input Nodes

Node N acts as an input node to circuit C when the signal applied to N by the "outside world" overpowers the signal applied to N by C. Let T be the transistor gating the N and O be the node on the other side of the transistor from N. A node acts as input node when, for any time t, the net on N's side of the transistor (called *NNET*) overpowers the net on O's side of the transistor (called *ONET*). Therefore, a flow constraint for input node N has the form $P \Rightarrow$ overpowers($NNET_N, ONET_O$) where P is the predicate which causes *NNET* to be on one side of T and *ONET* to be one the other side of T. When flow constraints are generated *NNET* always has the form [N]. When flow constraints are propagated (*cf.* Chapter 11) *NNET* becomes a net.

For example, consider the Inverting Latch again. Its In's net behavior equation is

$$\begin{aligned} \mathrm{In}_{net} = \lambda\ t\ .\ & \mathrm{Latch}_b(t) \rightarrow [\mathrm{In\ S}], \\ & \mathtt{NOT}(\mathrm{Latch}_b(t)) \rightarrow [\mathrm{In}]. \end{aligned}$$

For In to be an input node $[\mathrm{In}]_{In}$ must overpower $[\mathrm{S}]_S$ whenever Latch$_b$ is true. Because of this, the constraint

$$\mathrm{Latch}_b \Rightarrow \mathtt{overpowers}([\mathrm{In}]_{In}, [\mathrm{S}]_S)$$

is generated. (This is the third constraint of the four constraints generated for the Inverting Latch.) In general, an input node having N clauses in its net behavior

[1]Silica Pithecus defines "sufficiently greater" as an order of magnitude greater.

expression will generate $N-1$ flow constraints. The following algorithm generates flow constraints for input nodes.

Algorithm Generate Flow Constraints for an Input Node *I*:
Inputs: *I*'s net behavior *NB*.
Outputs: The flow constraints ensuring *I* acts as an input node.
Method:

```
For each clause $C$ of $NB$ do
  Let $Net$ be the consequent of $C$ and $P$ be the predicate of $C$.
  Let $T$ be the transistor gating $I$.
  Let $O$ be the node on the other side of $T$ from $I$.
  Let $ONET$ be the net on $O$'s side of $T$.
    Issue the flow constraint "$P \Rightarrow$ overpowers$([I]_I$, ONet$_O)$". ■
```

5.1.2 Output Nodes

Node N acts as an output node of circuit C only when information flows out of C through N. This is the inverse of an input node. Using the same notation as for flow constraints on input nodes, flow constraints on output nodes look like $P \Rightarrow \mathtt{overpowers}(\mathrm{ONET}_O, \mathrm{NNET}_N)$.

Output constraints are not explicit. When an output node connects with an input node, satisfying the flow constraints of the input node automatically satisfies the flow constraints of the output node. When two or more output nodes connect, Silica Pithecus checks for conflicts: when two different devices can simultaneously drive the same node to different values an error is signalled.

5.1.3 Buses

Besides input and output nodes, circuits also use *bus nodes*. A bus node sometimes acts as an input node, and sometimes acts as an output node. The methods above do not work for bus nodes because they are not purely either input or output nodes. The solution is to examine the intended digital behavior to determine when a bus node is to act as an input and when it is to act an output. Flow constraints are then generated to make buses act appropriately.

5.2 Timing Constraints

When a net *NET* contains an input node special care must be taken to ensure the input node doesn't corrupt *NET*. If the input node changes value before being shed from *NET* then *NET* may be corrupted. Extra timing constraints (specifically, `falls-first` constraints) must be generated to avoid this form of corruption. The following algorithm is used to generate these constraints.

Algorithm Generate More `falls-first` Constraints:

Inputs: The net transition graph G of node N.

Outputs: The `falls-first` constraints which prevent Input Node Corruption of N_s.

Method:

```
For each vertex $V$ in $G$ do
  Let $P$ be the predicate choosing $V$.
  If $V$ contains an input node $I$ then
    For each downward outgoing edge $D$ from $V$ to $W$ do
      Let $SD_s$ be the signal causing $D$.
      If $W$ does not contain $I$ then
        Issue the constraint "$P \wedge \neg SD_b(t_f) \Rightarrow$ falls-first($SD_s, I_s$)." ■
```

Chapter 6

Silica Pithecus

Silica Pithecus is the implementation of my theory of multilevel verification of MOS circuits. This chapter discusses the structure of Silica Pithecus and presents parts of Silica Pithecus undeserving their own chapter.

The major operations performed by Silica Pithecus depend on whether it is performing flat verification or hierarchical verification. The primary operations of Silica Pithecus when performing flat verification are constructing a representation of signal behavior that makes invalid signals apparent, and issuing constraints which ensure those invalid signals never arise. The primary operation of Silica Pithecus when performing hierarchical verification is processing constraints to show they hold.

Common to both forms are the inputs to Silica Pithecus, namely the specification of digital behavior and the specification of circuit structure. Digital behavior is specified as a program written in a programming language supplied by Silica Pithecus. This language has an interpreter, specifications can be debugged by executing them. Chapter 7 discusses in detail the specification of digital systems and the motivation for Silica Pithecus's method of specifying digital behavior. The specification of circuit structure is not presented in this dissertation, but the representation of circuit structure is.

This chapter has three sections. The first section discusses the structure of Silica Pithecus. The representation of circuits is presented in the second section. The last section describes how Silica Pithecus compares digital formulas.

6.1 The structure of Silica Pithecus

The structure of Silica Pithecus depends upon the type of verification it is performing. We first discuss Silica Pithecus's operation during flat verification, then its operation during hierarchical verification.

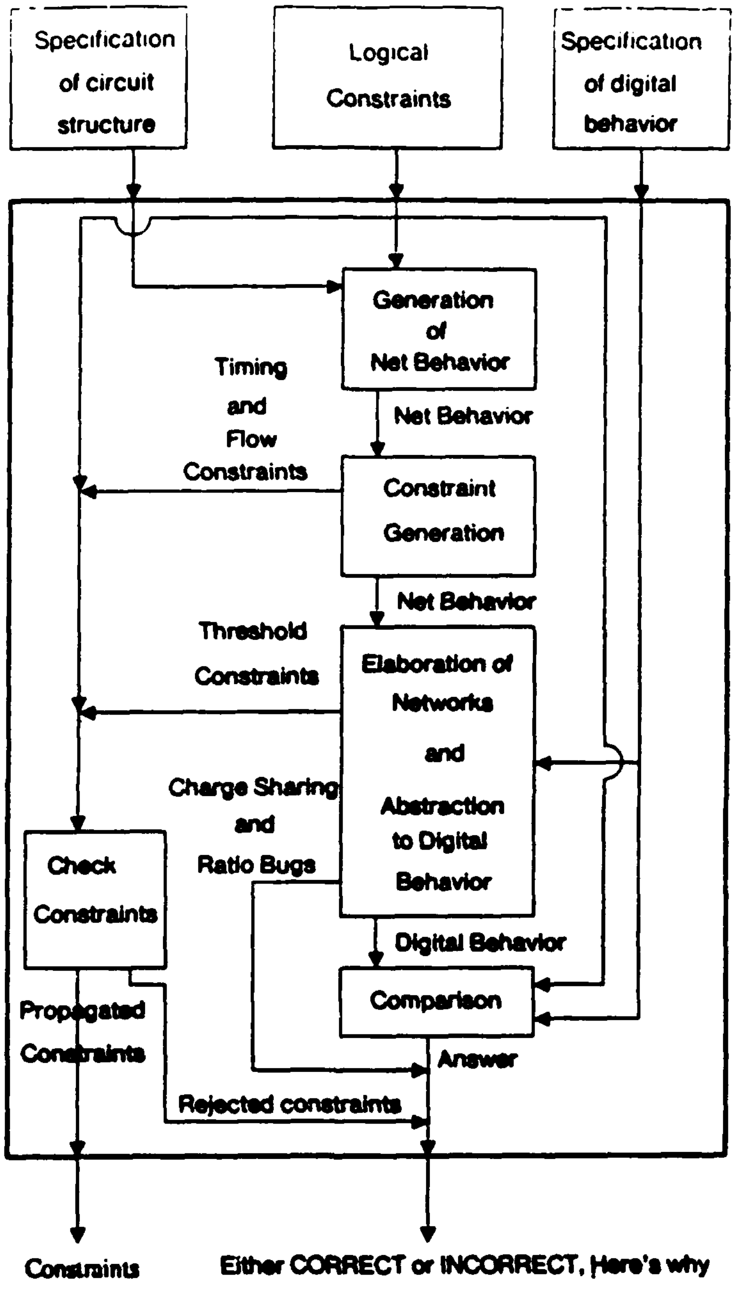

Specification of circuit structure
Logical Constraints
Specification of digital behavior
Generation of Net Behavior
Timing and Flow Constraints
Net Behavior
Constraint Generation
Net Behavior
Threshold Constraints
Elaboration of Networks and Abstraction to Digital Behavior
Charge Sharing and Ratio Bugs
Check Constraints
Digital Behavior
Comparison
Propagated Constraints
Answer
Rejected constraints
Constraints
Either CORRECT or INCORRECT, Here's why

Figure 6.1: Outline of Silica Pithecus When Performing Flat Verification.

When performing flat verification Silica Pithecus has five phases (Figure 6.1). They are:

Generation of net behavior. Net behavior, introduced in Chapter 2, is a representation of a circuit's dynamic structure. Net behavior makes explicit the timing constraints needed to ensure state is not corrupted. A hierarchical method is used to generate net behavior. Chapter 8 presents the details of generating net behavior.

Issuance of timing and flow constraints. The algorithms for generating timing and flow constraints were presented in Chapters 4 and 5.

Generating and abstracting signal behavior. Silica Pithecus generates signal behavior from net behavior. Signal behavior is then abstracted to the digital level. During this phase threshold constraints are generated.

Although generating signal behavior from net behavior is conceptually separate from abstracting digital behavior from signal behavior, the two are discussed together. This is an artifact of intentional analysis; signal behavior itself is not important. The abstraction of signal behavior is. Therefore, much of signal behavior generation is bypassed,[1] which forces us to treat the two operations, generation and abstraction, together. Chapter 9 presents this part of Silica Pithecus.

Checking the generated constraints. The generated constraints are processed. Rejected constraints are returned to the designer as errors. Propagated constraints are attached to the circuit being verified. (At any given level of structural hierarchy a constraint is either satisfied, violated, or propagated to the next higher structural level.)

Comparing arbitrary digital expressions. The comparison of digital expressions is a time-consuming operation. The general problem is NP-Complete. Silica Pithecus does not try to solve this problem elegantly. Instead, it relies on hierarchy to manage the problem. Silica Pithecus's comparison methods are presented in the third section of this chapter.

The general theory of hierarchical verification is presented in Chapter 10. Chapter 10 shows the advantages of hierarchical verification, states the requirements on a verification system that make it work, and gives some concrete examples in the arithmetic domain.

When performing hierarchical verification Silica Pithecus has four phases (Figure 6.2):

[1] Which is good, since generating actual signal behavior is very hard!

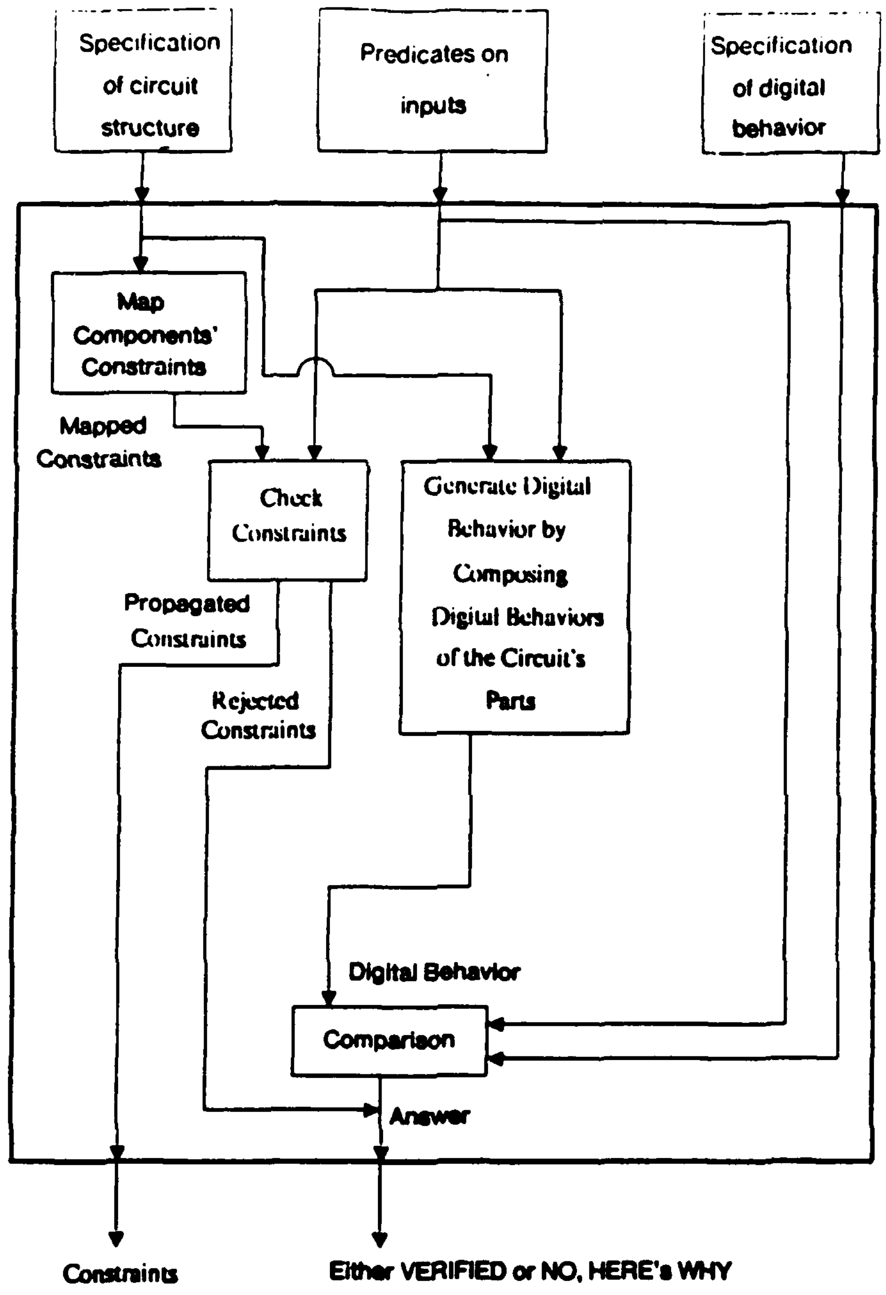

Specification of circuit structure
Predicates on inputs
Specification of digital behavior
Map Components' Constraints
Mapped Constraints
Check Constraints
Generate Digital Behavior by Composing Digital Behaviors of the Circuit's Parts
Propagated Constraints
Rejected Constraints
Digital Behavior
Comparison
Answer
Constraints
Either VERIFIED or NO, HERE's WHY

Figure 6.2: Outline of Silica Pithecus When Performing Hierarchical Verification.

PARTS

Name	Type	Renaming Information
STORAGE-GATE	*Transistor*	Drain ≡ A, Source ≡ Gnd, Gate ≡ Storage
READ-GATE	*Transistor*	Drain ≡ Bus, Source ≡ A, Gate ≡ Read
WRITE-GATE	*Transistor*	Drain ≡ Bus, Source ≡ Storage, Gate ≡ Write

CONNECTIONS

Bus	(Drain READ-GATE) (Source WRITE-GATE)
A	(Source READ-GATE) (Drain STORAGE-GATE)
Storage	(Gate STORAGE-GATE) (Drain WRITE-GATE)
Unnamed	(Source STORAGE-GATE) Gnd

Table 6.1: Representation of the Three Transistor Ram

Mapping components' constraints. The major part of hierarchical verification is the mapping of the five different types of constraints. Each type of constraint has its own special handler. Chapter 11 discusses constraint mapping and constraint checking.

Checking the mapped constraints. This the same as for flat verification.

Generating digital behavior. The digital behavior of the whole is the composition of the components' digital behaviors.

Structural comparison. The structural hierarchy of the derived digital behavior is compared against the funcational hierarchy of the intended behavior. The comparison is described below.

6.2 The Hierarchical Representation of Circuits

Circuits are specified hierarchically where the primitive devices are those provided by tl : technology being verified (*e.g.*, DFETs and EFETs in nMOS). My representation for circuit mirrors and exploits this hierarchy. This representation is nearly identical to DPL's [DPL].

The primitive circuit element is the transistor. It has three terminals called Drain, Source, and Gate. Consider the Three Transistor Ram (Figure 8.2), which is composed solely of transistors. The representation of this circuit consists of a listing of its parts and the connections between them (Table 6.1). Circuit names (*e.g.*, *Transistor* or *Three Transistor RAM*) are written in italics, and instance names (*e.g.*, STORAGE-GATE) are hyphenated and capitalized.

The structural representation of a circuit has two sections. The Connections section lists and names the connections between the components. The Parts section lists the components, their names, and *renaming information*. The renaming information dictates how names of components map to names of the parent circuit. Renaming information is used heavily. Note that the component's names (*e.g.*, *transistor*) are listed, not the components themselves. This is a "pointer" scheme by which only one *prototype* of a circuit exists regardless of how many larger circuits it is part of. There are two advantages to this scheme. The first is a saving in storage for circuits used many times. The second, and more important, is having to analyze a circuit once instead of the number of times it occurs in larger circuits.

6.3 Comparison

The type of comparison performed depends on whether flat or hierarchical verification is performed. Flat verification requires comparing arbitrary digital expressions for equality. Arbitrary comparison is computationally intensive. Hierarchical verification requires comparing structurally similar digital expressions. This type of comparison is computationally inexpensive.

Silica Pithecus uses a very simple theorem prover to show two digital expressions are equivalent. The only complication is the presence of *float*'s which occur in conditional expressions.[2] When faced with conditional expressions C1 and C2 Silica Pithecus shows that they are equivalent assuming that the first predicate of C1 is true and assuming that the first predicate is false. This generates two simpler subproblems. The second subproblem may involve conditional clauses, in which case the strategy is reapplied.

When hierarchical verification is performed the structural hierarchy of the schematic and the functional hierarchy of the intended behavior must match. Silica Pithecus determines if the input/output relationships of the functions in the digital specification and the wires in the structure correspond (*i.e.*, the procedures and components are similarly "wired").

For example, consider a sum of products device (Figure 6.3). Let the intended behavior of this device be

```
(db (Sum-of-products x y z)
  (+ (* x y) (+ (* x z) (* y z))))
```

This device is correct only if devices PLUS1 and PLUS2 were verified to have digital behavior '+", devices TIMES1, TIMES2, and TIMES3 were verified to have behavior *, and the are wire according to the dataflow prescribed by the intended behavior of the entire device.

[2] *Don't care* values would cause similar problems.

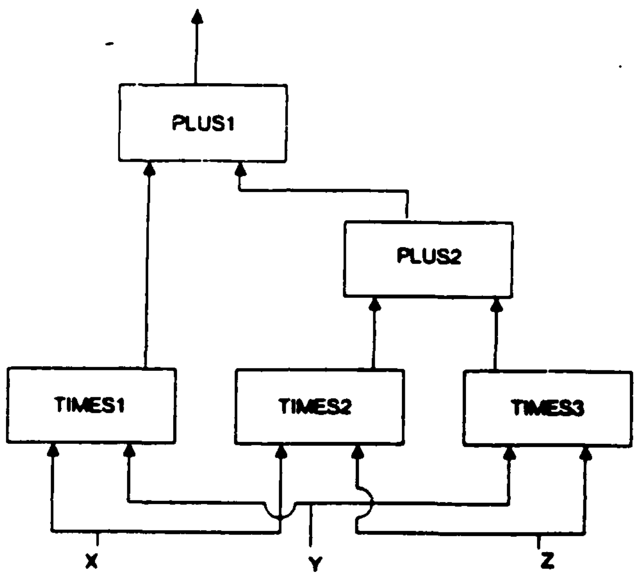


Figure 6.3: Sum of Products

Chapter 7

Specifying Digital Systems

As defined in Chapter 3, the digital descriptive level consists of Booleans, *float*, boolean functions, and join. All complex digital systems are created from these primitives. This chapter addresses how we specify the combination of these primitives to build complex devices.

Digital systems are hard to describe because there is no single notation that works well for small (*e.g.*, an adder cell), medium (*e.g.*, a 16 bit adder), large (*e.g.*, a controller), and very large (*e.g.*, a whole chip) designs. Each level has its own particular problems and nuances. Notations that work well for small designs are often too cumbersome for large designs and *vice versa*. The notation used by Silica Pithecus errs on the side of easily describing large designs at the expense of not concisely describing small designs.

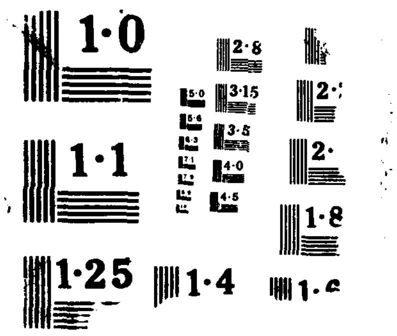
This chapter has five sections. The first section discusses the issues of describing digital behavior common to all designs regardless of size. The second section proposes using programs for specifying digital behavior[1] and covers the advantages and disadvantages of doing so. The third section gives a rationalization and overview of the applicative programming language used by Silica Pithecus for describing digital systems. The fourth section is a reference manual of this language. The last section presents a large example which shows the specification of a reduced instruction set computer.

To a large degree, there is nothing new in this chapter. Many of the ideas presented here are germane to the applicative programming community. Many of our arguments for using applicative programming languages in the description of digital systems appear in Johnson [Johnson] (though in a much less accessible form). Other researchers, [Sheeran, Johnson], use the same method of specification as we are about to propose; the only difference is usually syntax. Nonetheless, for completeness, and for the edification of readers unfamiliar with the applicative

[1]There is another camp which uses logic for specifications. Interested readers are referred to [Gordon84b], [Mishra/Clarke], and [Mosskowski].

AD-A195 489 FORMAL HIERARCHICAL MULTILEVEL VERIFICATION OF SYNCHRONOUS MOS VLSI DESIGNS(U) MASSACHUSETTS INST OF TECH CAMBRIDGE MICROSYSTEMS RESEARCH CENTER D W WEISE 2/2

UNCLASSIFIED NOV 87 VLSI-MEMO-87-425 F/G 9/1 NL


1·0
2·8
3·15
5·0
5·6
3·6
4·0
4·5
2·
2·
1·1
1·8
1·25
1·4
1·6

approach, we now present a thorough discussion of the subject.

7.1 Issues in Specifying Digital Behavior

The major issues in specifying digital behavior are *coverage*, *extent*, and *reliability*.

Coverage refers to specifying the behavior of a circuit for all its inputs and states. A specification that does not completely cover all inputs and states is called *incomplete* or *partial*. As will be discussed in more detail below, it is hard for specifications to exactly cover the intended digital behavior of a circuit: programs tend to over specify behavior, and logic tends to under specify behavior.

Extent refers to the interval of time (measured in events, such as clock ticks, not real time) used in the specification. For example, one may wish to specify that once asserted signal X_s remains high until signal Y_s goes low; this specification has greater extent than one time unit.

Reliability measures the amount of faith one has that a specification is correct.

As a basis for later comparison consider the *listing method* of specification: behavior, *i.e.*, next state and outputs, is specified by listing for each input and state of interest what the intended behavior is. (This method is, of course, infeasible due to the exponential number of entries required for even a modest circuit.) Assuming the listing was done correctly the coverage would be exact, there would be no spurious or lacking entries. It would have no extent. Its reliability could be checked by writing a simulator for the listing and running the listing as if it were a program.

7.2 Specifying Digital Behavior with Programs

Instead of listing all the entries one can write a program in a suitable language to compute the intended behavior. I define a "suitable language" as being small with well defined semantics. This section gives the advantages and disadvantages of specifying digital behavior with programs.

7.2.1 Advantages of Using Programs

Programs Can be Debugged and Verified

Because a program can be executed and debugged, it can be very reliable. In comparison to text editors and spreadsheets, the programs describing the digitial behavior of even million transistor chips are rather small and extremely heavy testing of the program can occur. It is also possible to verify programs. Indeed, proving the functional correctness of digital designs is essentially the same problem as proving programs correct.

Programs Allow Hierarchical Specifications

An important property of programs is that they can be composed to make larger programs. When functional programming languages are used functional composition alone suffices for composing programs. Programs therefore allow for hierarchical and modular specifications. The specification's hierarchy can be exploited to great effect during verification if it matches the structural hierarchy of the design being verified (this is discussed in Chapter 6).

7.2.2 Disadvantages of Using Programs

Programs are Poor at Specifying Asynchronous Systems

Programs can be used to *describe* asynchronous systems. For example, CSP [reference] and ADA tasks [Reference] can be used to models asynchrnous systems. Nonetheless, programs fail for specifying the intended behaviors of systems. The problem is that what we often wish to prove some property of the behavior holds. That after some time, the system will acknowledge a request, or that some arbiter properly arbitates. Such properties are best specified by temporal logics [Mishra/Clarke] rather than by programs.

Programs Have Coverage Problems When Inputs are Not Restricted

Programs are prone to have coverage problems, they often specify behavior that should be left unspecified (*i.e.*, a value is specified where there should be a *don't care*). This "bug" is very hard to catch unless it is specifically tested for: normal tests of program will probably never exercise the code that is "buggy." Indeed, the program may have all the functionality and work correctly for all correct inputs. But the circuit to be verified may assign different values to the don't cares. When verified against the program it would not pass because it did not do what the program did.

A concrete example will make this clear. Consider a memory circuit for storing four bits. There is one bus, four read lines, and four write lines. The designer may intend for only one bit to be writeable at a time and design the circuit that way. That is, the circuit will write invalid values if more than one write line is asserted. A program to specify the memory could be written and "debugged" with only one write line asserted at a time without the designer noticing the program allows simultaneous writes to more than one bit. Even though the circuit may be correct and the specification "debugged," the circuit cannot be verified against the program since the circuit does not allow more than one bit to be written whereas the program does.

The real bug is that the verifier does not know the constraints on the input signals. If they are known, a verifier would not attempt to verify behaviors arising from disallowed input configurations (*e.g.*, more than one write line asserted at a time). Silica Pithecus, which uses programs as specifications, requires constraints on inputs to be explicitly declared. These constraints are used during the comparison of the abstracted and specified behaviors.

7.3 Rationale for Silica Pithecus's Specification Language

The specification language should have several attributes:

- It should be a programming language with an interpreter.
- It should support modular specifications.
- It should have simple and precise semantics.
- It should be able to express parallelism.

This section discusses these goals and summarizes the language resulting from satisfying them.

7.3.1 The Goals of the Language

It should be a programming language with an interpreter.

One should be able to debug the specifications by running them. If the specifications are not correct then the design will not be.

It should support modular specifications.

Modularity is the mechanism used to contain complexity. Complex systems are built by composing together less complex systems. A specification language must support modularity by allowing specifications to be composed to make complex specifications. For example, a specification for a control unit and a specification for an ALU should be composable using functional composition. If either specification must be changed in order for them to be composed, then the language does not support modular specifications. Under this restriction, languages such as Pascal could not be used as a specification language because of potential clashes between names of variables.

This goal constrains the evaluation mechanism for the language. Consider again composing a controller and an ALU together. If they simultaneously transmit data to each other then they cannot be emulated by standard applicative order:

that would require evaluating one before the other preventing simultaneous communication. Some form of lazy evaluation must be used to model simultaneous communication.[2]

It should have simple and precise semantics.

Simple and precise semantics is required by the need to know what a program in the language means. If that is not easily known then comparing programs (*i.e.*, the abstracted and specified digital behaviors of a circuit) is meaningless.

It should be able to express parallelism.

The ability to express parallelism is mandatory because a digital circuit may be computing many things simultaneously.

7.3.2 Overview of the Resulting Language

The language that resulted from these goals is a functional language whose data types include booleans, *floats* (whose sole member is *float*), *vectors* (ordered collections of objects), *streams* (infinite lists of objects), and both simple and higher order procedures. Certain binding forms automatically delay the evaluation of their arguments.

All of the goals are met by virtue of the language being functional and having first class function objects. A functional language does not specify the order of evaluation of its terms. It therefore supports expressing parallism. Functional langauges generally have simple and precise semantics, this one has so few data types it's simpler than most.

The only problem with this approach is specifying the behavior of circuits containing state: side effected variables cannot be used to model changing state. There are two solutions to this problem. The first treats circuits as functions from inputs and states to outputs and new states. This solution is undesirable because of the notational overhead of explicitly recapturing and transmitting state in every function which specifys or uses an object with state. The second treats circuits containing state as functions taking the initial state of the circuit and returning a function that takes a stream of the circuit's inputs over time and returns a stream of the circuit's outputs over time. Using this convention, purely combinational circuits are specified as functions mapping streams of input to streams of output. We will use the second solution.

[2]Although there are primitives in the language for creating lazy constructs (*i.e.*, lambda), the more perspicuous descriptions of digital systems result when lazyness is explicitly provided by the language.

By using streams, functional composition can be employed to compose specifications of objects containing state. For example, assume that A, B, and C are single bit shift registers already initialized with state, and that they take two arguments, Input and Shift. Both arguments are streams of 1's and 0's. When a component of Shift is 1 the corresponding components of Input is shifted in and the current state shifted out; otherwise the shifter maintains its current state. A three bit shift register (*i.e.*, it takes three shift instructions for the input to reach the output) can be specified as

```
(lambda (input shift) (A (B (C input shift) shift) shift)).
```

7.4 The Digital Specification Language

This section is a detailed description of the language used by Silica Pithecus. It is heavily modelled after Scheme [Abelson].[3] It differs in that it lacks certain data types and features (no catch/throw or quoting mechanism), and it adds destructuring binding and new function defining forms. It uses lexical scoping and a mixture of lazy and applicative evaluation. We do not present a complete description of the language. Only a listing of the data types, the functions operating on those types, and the *special functions*[4] of the language are presented. Readers not interested in the details can safely skim this section and the next section.

7.4.1 Data Types

There are five data types: Booleans, Floats, Vectors, Streams, and Procedures.

Booleans

There are only two boolean values *True* and *False*. They are denoted **1** and **0**, respectively. The standard boolean functions **and**, **or**, and **not** are primitively provided. **And** and **or** take an unlimited number of arguments.

Floats

There is only one floating value, *float*. It is bound to the identifier **float**. The only function in the language which accepts float is **join**. **Join** takes an arbitrary number of arguments. Each argument must be either *float*, *true*, or *false*. If all the arguments are *float* the result is *float*; if all the non *float* values are *true* (*false*) the result is *true* (*false*). *Float* is used to model tri-state busses.

[3]Modulo syntax, my specification method for circuits with states is very similiar to the treatment of objects with states in Section 3.4.4 of [Abelson].

[4]This term corresponds to what others have called *special forms*.

Vectors

Vectors are a data aggregating mechanism. They act like Lisp's conses. The primitive vector, which has no elements, but is bound to the identifier ***empty***. A vector of length N is created using the procedure **vector-cons** which accepts an object of any type and a vector of length $N-1$, and returns a vector of length N. If **z** is **(vector-cons a b)** then **(first z)** is **a** and **(rest z)** is **b**. The function **nth-vector** accepts a vector and an integer. It returns the nth element of the vector where the counting is zero-based. The function **new-vector** accepts a vector, an integer, and an object. It creates a new vector whose elements match the original except for the nth one, which is the object.[5] The function **vector** takes an arbitrary number of arguments and returns a vector containing the arguments. That is **(vector a b c d)** is the same as **(vector-cons a (vector-cons b (vector-cons c (vector-cons d *empty*))))**. **Vector** can be abbreviated **v**.

#<n>v<m> denotes a function call that returns a vector whose contents, when interpreted as binary representation of a number, is **m**. **<n>**, which is optional and defaults to 8, is the length of the vector the function creates. For example, typing **#3v4** is the same as typing **(vector 0 0 1)**.[6]

The function **=** when applied to vectors returns TRUE if the vectors have the same length and if the corresponding elements of the vectors have the same boolean values.

Streams

A stream is a mechanism for representing "infinite vectors." A stream is constructed with **stream-cons** which accepts two arguments. If **z** is **(stream-cons a b)** then **(now z)** is **a** and **(future z)** is **b**. **Stream-cons** is like **vector-cons**, however, the second argument, rather than being evaluated, is *delayed*. When **future** is called to retrieve the second argument it is *forced* to yield a value. By this mechanism delayed expressions can make use of variables which are defined after **stream-cons** is called. **Stream-cons** can be abbreviated **sc**.

The function **stream-map** is used to map a function across a stream. For example, let **x** be a stream of boolean values. **(stream-map (lambda (y) (not y)) x)** returns a stream whose values are the logical negations of **x**'s values.

Procedures

Procedures are first class objects created by **lambda**. For example,

```
(lambda (x y z)
```

[5]The implementation need not copy the whole vector as long as it acts like it does.

[6]By my conventions the first bit in a vector is the low order bit.

```
  (and x (or y z)))
```

denotes the function that accepts three arguments and returns the logical-and of its first argument with the logical-or the other two arguments.

Notes on Types

There are no functions for determining the type of an cbject.

7.4.2 Special Functions

A *special function* is a function not all of whose arguments are evaluated before invoking the function. Two special functions, **cons-stream** and **lambda**, have already been introduced. There are four types of special functions: special functions for controlling the evaluation process, special functions for creating procedures, special functions for binding variables, and special functions for naming "top-level" objects. The four types are exemplified by the special forms **cond**, **lambda**, **let**, and **define**, respectively.

Evaluation of Control Special Functions

There are three special functions for controlling evaluation: cond, if, and select. The syntax of if is (if <predicate> <consequent> <else>). First if evaluates its predicate. If the predicate returns *True* then if evaluates its consequent. Otherwise if evaluates its else clause.

Cond is a generalization of if. Its syntax is (cond (<p1> <c1>) (<p2> <c2>) ... (<pn> <cn>)). It evaluates the p's in turn. When one returns *True*, cond evaluates its associated consequent. If any predicate returns a non-boolean value or no predicate returns *True* an error is signaled.

The syntax of select is

```
(select <selector>
  (<c1> <exp1>)
  (<c2> <exp2>)
  ...
  (<cn> <expn>))
```

This is the same as writing

```
(cond ((= <selector> <c1>) <exp1>)
      ((= <selector> <c2>) <exp2>)
      ...
      ((= <selector> <cn>) <expn>)).
```

Procedure Creating Special Functions

The only special function for creating procedures is lambda. Its syntax is

```
(lambda (<arg1>, ..., <argn>) <exp>)
```

This create a procedure of n arguments whose body is <exp>. All variables are lexically scoped.

Variable Binding Special Functions

There are two special functions for binding forms: let and rlet. The syntax of let is

```
(let ((<name1> <exp1>)
      (<name2> <exp2>)
      ...
      (<namen> <expn>))
  <body>).
```

Let evaluates all the exps and then binds them to the names and then evaluates <body>. Names are either a sequence of characters, as in X, Y, and this-is-a-variable-name, or a list of character sequences enclosed in parenthesis such as (A this-is-name C). In the latter case, the associated expression must return a vector the same length as the list. Each name in the list is bound to the corresponding element of the vector. This is called *destructuring binding*. Destructuring binding is used to work around the restriction that procedures only return one value. Examples are provided below.

Rlet stands for Recursive Let. It is like let except that the expressions of the rlet can refer to the variables being bound by the rlet and the evaluation of the expressions are delayed until they are used. This allows simultaneous communication between components of a specification.[7]

For example, assume that Datapath and Controller are procedures representing their namesakes. Assume further that each takes two inputs, one from the outside world and one from the other device, and that they each produce two outputs. They would be connected together as follows:

```
(rlet (((controller-to-datapath-wire controller-output)
        (controller controllers-input datapath-to-controller-wire))
       ((datapath-to-controller-wire datapath-output)
        (datapath datapath-input controller-to-datapath-wire))
  (vector controller-output datapath-output)))
```

[7]Unfortunately, this mechanism interacts poorly with destructuring binding. Pathological circuits can be specified whose specifying programs won't run due to definitional loops. To get around this problem, the language must really be completely call-by-need.

The output of this expression is a vector containing the outputs of the controller and datapath which are destined for the outside world. This example uses destructuring binding to get at the two outputs of each subdevice (which each return a vector of their two output). A more concrete and fleshed out example is presented below.

Special Functions for Naming Objects

Define is used for naming objects at "top-level." The syntax is (define <name> <exp>). For example, (define x 3) associates X with the value 3. Similarly (define foo (lambda (x y) (+ x x y y))) associates the procedure created by lambda with the name foo.

There are many syntactic sugarings for naming functions. These syntactic sugarings are motivated by the desire to make functions which operate on streams easy to write.

Df (for Define Function) creates a procedure and associates the procedure with a variable. The syntax is

```
(df (<name> <var-list>) <body>).
```

This is the same as

```
(define <name>
  (lambda (<var-list>)
    <body>)).
```

Db (for Define Behavior) is used for specifying the behavior of circuits without state. It lets one write a function apparently over scalars (*i.e.*, non streams) but which operates over streams. For example, whereas the function defined by

```
(df (xor a b)
  (and (or a b) (not (and a b))))
```

takes scalars,

```
(db (Xor a b)
  (and (or a b) (not (and a b))))
```

defines a function taking and returning streams. Assuming only one argument (db (<name> <arg>) <body>) is the same as

```
(define <name>
        (lambda (<arg>)
          (stream-map (lambda (<arg>) <body>) <arg>)))
```

The equivalences for functions having more than one argument are extrapolated from this one. I use the typographic convention of capitalizing names of functions which map streams to streams (or which map vectors of streams to vectors of streams).

The behavior of circuits with state is specified using a syntactic variant of **df**. Like **db**, it transforms functions taking scalars into functions taking streams. For example, consider a shift register which has one bit of state. It continuously reads a value and outputs the previous value read. It is specified

```
(df ((SHIFTER state) input)
  (stream-cons state (shifter input))).
```

This binds **shifter** to a function taking an initial value and returning a function that maps a stream into a stream. Using plain **df** this would be specified as

```
(df (SHIFTER state)
  (lambda (input)
    (stream-cons
      state
      ((shifter (now input)) (future input))))).
```

This is a simple example. For a more detailed circuit, such as a register with many inputs, the overhead of continually writing, for example, **(now input1)** and **(future input2)**, is very large. I use the typographic convention of writing names of functions which take state information and then return a function mapping streams to streams (or vectors of streams to vectors of streams) in uppercase.

Assuming one argument,

```
(df ((<name> <state>) <input>)
    <body>)
```

is the same as

```
(define <name>
  (lambda (<state>)
    (lambda (<input>)
      (map-function-stream
        (lambda (<input>) <body>)
        input))))
```

where **map-function-stream** is

```
(df (map-function-stream function-making-a-stream input-stream)
  (let ((stream
          (function-making-a-stream (now input-stream)))
```

```
    (stream-cons
      (now stream)
      (map-function-stream
        ((future stream) (future input-stream))))))
```

7.5 Examples

I will specify the digital behavior of a Fanatically Reduced Instruction Set Computer (FRISC).[8] A picture of the machine and its architecture appears in Figure 7.1. The computer has the same interface and instructions as Gordon's example computer [Gordon83],[9] but is implemented as a RISC machine (*i.e.*, every instruction executes in one cycle) rather than as a microcoded machine. I also specify its behavior in much more detail. For example, Gordon treats the ALU and memory of his computer as primitive, whereas I fully specify the them. The difference in detail is due to his concern for verifying the functional behavior of the computer (*i.e.*, prove its microcode is correct), whereas I wish to verify the digital behavior of the computer.

To the user the machine has a 13 bit Program Counter (PC), a 16 bit accumulator, a 2^{13} word memory where a word is 16 bits wide, and 8 instructions. The instructions are

HALT: Stop executing the program.

JMP address: Set PC to *address*.

JZRO address: If the contents of the accumulator is zero then the PC is set to *address*.

ADD address: Increment Accumulator by the contents of *address*.

SUB address: Decrement Accumulator by the contents of *address*.

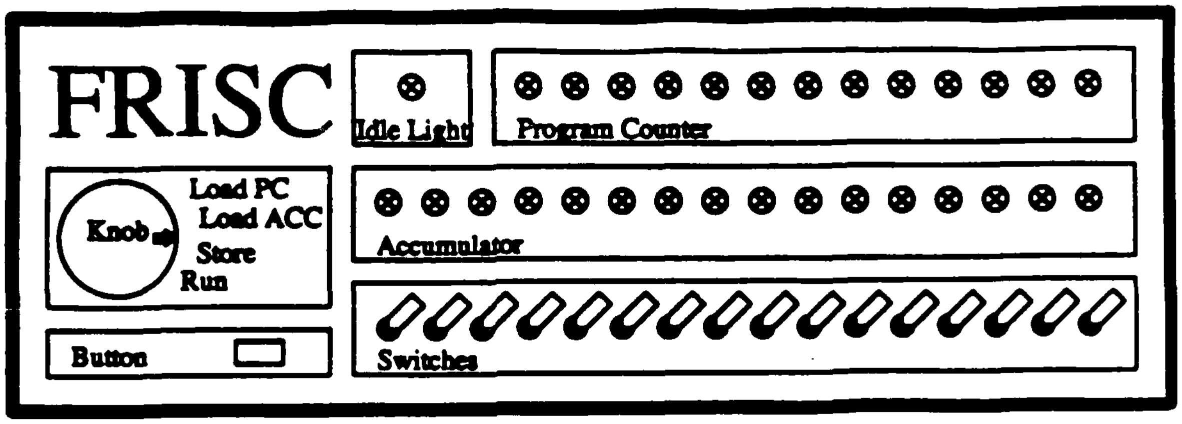
LD address: Load Accumulator with the contents of *address*.

ST address: Store Accumulator into *address*.

NOP: Do nothing.

[8]This title is due to Jeff Siskind [Siskind].

[9]Barrow also uses Gordon's example computer in Verify [Barrow]. Everything I say about Gordon's example computer apply to Barrow's example computer.



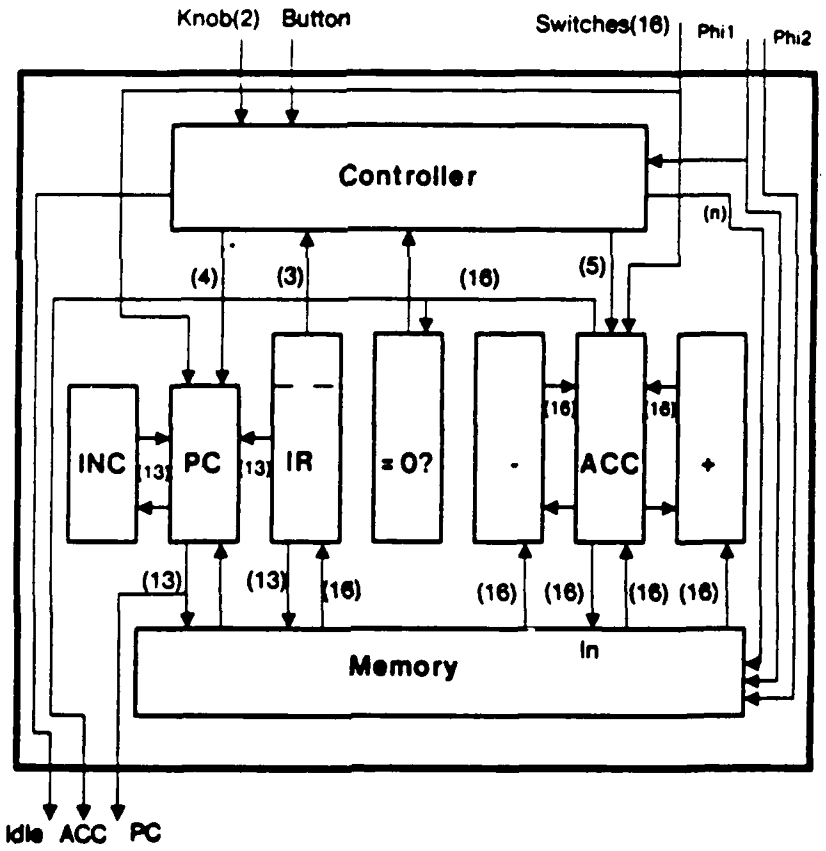


Figure 7.1: A Fanatically Reduced Instruction Set Computer. This computer has 8 instructions each of which takes one cycle to execute. Its word size is 16 bits and its address space is 13 bits.

The computer's front panel contains a bank of 16 toggle switches, lights displaying the contents of the PC and the Accumulator, an idle light displaying the computer's state, a four position knob, and a button. If the machine is running pushing the button causes it to stop running. The knob determines what happens when the button is pushed when the machine is idle as follows:

0 The PC is loaded from the toggle switches.

1 The Accumulator is loaded from the toggle switches.

2 The memory location specified by the switches is loaded from the accumulator.

3 The program starts running from the location in PC.

I will specify the computer in four steps. Each step illustrates a different aspect (and idiom) of specifying digital designs. The first step specifies cells. It emphasizes that a device's inputs and outputs are streams, not single boolean values. The second step groups cells together. It highlights the creation of vectors and the operations on vectors. The utility of using recursive functions to create arbitrarily sized devices is also discussed. The third step specifes the controller and datapath. It shows how large complex devices containing many disparate parts are wired together. Finally, the entire computer is specified. This is specification is amazingly small (as the reader can verify by peeking ahead to the end of this section).

7.5.1 Small Cells

I will show the specification of a combinational circuit (an adder cell) and a circuit containing state (a register cell).

The Adder Cell

A full adder takes three inputs and produces two outputs (Figure 7.2). The specification of this circuit uses a helper function called **xor**.

```
(df (xor a b)
  (and (or a b) (not (and a b))))

(db (Adder-cell a b c)
  (vector (xor a (xor b c))
          (or (and a b) (and a c) (and b c))))
```

Because **db** was used to define the adder, its inputs and outputs are streams.

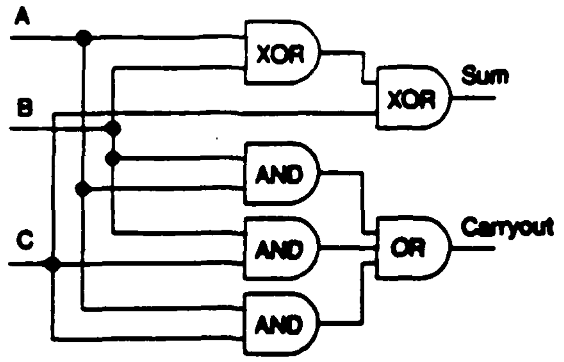


Figure 7.2: A Full Adder. This is a pictorial representation of the digital behavioral specification of an adder. This is not the MOS implementation.

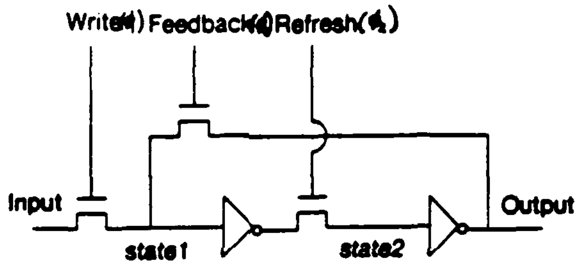


Figure 7.3: A Register Cell. On ϕ_1 it is fed with data which is either fed back from its output or fed from an external source. On ϕ_2 the data is passed to the output gate.

The Register Cell

The register cell operates in two phases. On the first phase it is loaded with data. This data can either be fed back from its output or be external. On the second phase the data is passed from the "input port" to the "output port." Figure 7.3 shows the nMOS implementation of this cell. Below is the digital specification of the cell.

```
(df ((REGISTER-CELL state1 state2) feedback refresh write input)
  (stream-cons
    ;; this is the output.
    (if refresh state1 (not state2))
    ;; this the next behavior of the device.
    (REGISTER-CELL (cond (write input)
                     (feedback (not state2)))
                 (cond (refresh (not state1))
                       (1 state2)))))
```

FRISC's PC has four input sources: feedback from itself, data from the memory, data from the instruction register (this implements jumps), and data from the incrementer. The register cell for the PC is specified as

```
(df ((PC-REGISTER-CELL state1 state2)
       feedback? refresh from-ir? ir-input
       from-memory? memory-input from-inc? inc-input)
  (stream-cons
    (if refresh state1 (not state2))
    (PC-REGISTER-CELL (cond (from-ir? ir-input)
                            (from-memory? memory-input)
                            (from-inc? inc-input)
                            (feedback? (not state2)))
                      (cond (refresh (not state1))
                            (1 state2)))))
```

7.5.2 Medium Circuits: Building Vectors of Cells

For the medium scale examples an adder which adds two similar length vectors together, and an arbitrary length register array will be specified. These circuits are specified as vectors of the appropriate cells. Because recursive functions are used in the specification, creating arbitrarily sized devices is automatic.

An N bit adder

In this example vectors will be used to represent unsigned integers. Given two equal length vectors (numbers) it produces their "sum." The zeroth element of a vector is the low order bit. The input vectors contain streams of booleans and the output of the adder is a vector containing streams of booleans.

```
(df (+ x y)
  (Adder-internal x y stream-of-0s))

(df (Adder-internal x y carryin)
  (cond ((and (empty-vector? x) (empty-vector? y))
         *empty*)
        ((or (empty-vector? x) (empty-vector? y)
         (error "ran out of one vector"))
        ;; the following let destructures the adder-cell's output
        (1 (let (((sum carryout)
                   (Adder-cell (first x) (first y) carryin)))
             (vector-cons
               sum
               (adder-internal (rest x) (rest y) carryout))))))
```

An N Bit Register

Like the adder, this specification works for arbitrary length inputs. The function **CREATE-N-BIT-REGISTER** is called with two vectors V1 and V2 each of length N. It then creates a register N bits long whose state is initialized to the contents of V1 and V2. The inputs to the register array are streams (for the control lines) and vectors of streams (for the word to be stored). Its output is a vector of streams.

```
(df (CREATE-N-BIT-REGISTER v1 v2)
  (let ((N-bit-register (CREATE-VECTOR-OF-REGISTER-CELLS v1 v2)))
    (lambda (feedback refresh write Input)
      (Operate-on-n-bit-register
        N-bit-register feedback refresh write input))))

(df (CREATE-VECTOR-OF-REGISTER-CELLS v1 v2)
  (if (empty-vector? v1)
      *empty*
      (vector-cons
        (REGISTER-CELL (first v1) (first v2))
        (CREATE-VECTOR-OF-REGISTER-CELLS (rest v1) (rest v2)))))

(df (Operate-on-n-bit-register Reg feedback refresh write Input)
  (cond ((and (empty-vector? Reg) (empty-vector? Input))
         *empty*)
        ((or (empty-vector? Reg) (empty-vector? Input))
         (error "ran out of a vector."))
        (1 (vector-cons
             ((first Reg) feedback refresh write (first Input))
             (Operate-on-n-bit-register
               (rest Reg) feedback refresh write (rest Input))))))
```

7.5.3 Large Circuits: the Controller and Datapath

The Controller

The controller has five inputs. Two of them, the knob, which determines what happens when the button is pushed, and the button, come from the front panel of the computer. Two, the opcode of the instruction being executed, and a signal asserting whether the contents of the accumulator is zero, come from the datapath. The last input is ϕ_1 which is supplied by a ϕ_1 generator living in the computer.

The controller has two outputs, a wire going to the idle light and a group of control signals going to the datapath. I will first give the constants used by the controller and then specify the controller.

```
;; these are the control signals

(define pc<-switches (v 0 0 0 1 0 0 0 0 0 0 0 1 0))
(define acc<-switches (v 0 0 1 0 0 0 0 0 0 0 0 0 1))
(define memory<-acc-no-fetch (v 0 0 1 0 0 1 0 1 0 0 0 1 0))
(define fetch (v 0 0 1 0 1 0 1 0 0 0 0 1 0))
(define nop (v 0 0 1 0 0 0 0 0 0 0 0 1 0))
(define pc<-ir (v 1 0 0 0 0 0 0 0 0 0 0 1 0))
(define pc<-pc+1 (v 0 1 0 0 0 0 0 0 0 0 0 1 0))
(define acc<-acc+mem (v 0 1 0 0 0 0 0 0 1 0 0 0 0))
(define acc<-acc-mem (v 0 1 0 0 0 0 0 0 0 1 0 0 0))
(define acc<-memory (v 0 1 0 0 0 0 0 0 0 0 1 0 0))
(define memory<-acc-fetch (v 0 1 0 0 0 1 0 1 0 0 0 1 0))
(define all-signals-low (v 0 0 0 0 0 0 0 0 0 0 0 0 0))

;; these are the opcodes

(define halt #3v0)
(define jmp  #3v1)
(define jzro #3v2)
(define add  #3v3)
(define sub  #3v4)
(define ld   #3v5)
(define st   #3v6)
(define nop  #3v7)

;; these are the states

(define idle #2v0)
(define fetch #2v1)
(define execute #2v2)
```

The computer can either be idling, fetching an instruction, or executing an instruction. Even when the computer is idling the controller is busy sampling the button on every cycle. If the computer is idle and the knob is in position 3 the computer will start running its program. If the computer is running then pressing the button will stop it.[10] The select operation is used to decode the state of the controller and the opcodes it receives from the data path.

```
(df ((CONTROLLER state) knob button =0? opcode phi1)
  (if phi1
    (select state
      (idle
        (if button
            (select knob
              (#2v0 (sc (v pc<-switches 1) (CONTROLLER idle)))
              (#2v1 (sc (v acc<-switches 1) (CONTROLLER idle)))
              (#2v2 (sc (v memory<-acc-no-fetch 1) (CONTROLLER idle)))
              (#2v3 (sc (v fetch 0) (CONTROLLER execute))))
            (sc (v nop 1) (CONTROLLER idle))))
      (fetch
        (if button
            (sc (v nop 1) (CONTROLLER idle))) ;halt
            (sc (v fetch 0) (CONTROLLER execute))) ;run
      (execute
        (select opcode
          (halt (sc (v nop 1) (CONTROLLER idle)))
          (jmp (sc (v pc<-ir 0) (CONTROLLER fetch)))
          (jzro (if =0?
                    (sc (v pc<-ir 0) (CONTROLLER fetch))
                    (sc (v pc<-pc+1 0) (CONTROLLER fetch))))
          (add (sc (v acc<-acc+mem 0) (CONTROLLER fetch)))
          (sub (sc (v acc<-acc-mem 0) (CONTROLLER fetch)))
          (ld  (sc (v acc<-memory 0) (CONTROLLER fetch)))
          (st  (sc (v memory<-acc-fetch 0) (CONTROLLER fetch))
          (nop (sc (v nop 0) (CONTROLLER fetch))))))
    (sc (v all-signals-low (= state idle)) (CONTROLLER state))))
```

[10]The button must be constructed to only give a single pulse on the first cycle it is held down. Otherwise the computer will thrash between the run and idle states.

The Datapath

The datapath contains 3 registers (PC, ACC, and IR), three arithmetic units (an adder, subtractor, and incrementer), a boolean tester (which returns *True* if the output of ACC is 0), and a 8K word memory of 16 bit wide words. The functions specifying the registers are similar to the function specifying the register array presented earlier in this section. The specification of the subtractor, incrementer, and boolean tester are similar to the specification of the adder. Only the memory unit is different, but its specification will not be presented here.

There are four outputs from the datapath. Two of them, the opcode of the instruction to be executed, and the zero test of the accumulator, are destined for the controller. The other two, the contents of the Program Counter and the Accumulator, are sent to the lights on the front panel.

```
(df (DATAPATH ir pc acc mem)
  (let ((Instruction-reg (CREATE-INSTRUCTION-REGISTER ir))
        (Program-counter (CREATE-PC-REGISTER pc))
        (Accumlator (CREATE-ACCUMULATOR-REGISTER acc))
        (Memory (CREATE-MEMORY mem))
    (lambda (control-lines phi1 phi2 switches)
      (let (((pc<-ir pc<-pc+1 pc<-pc pc<-sw
              ir<-mem
              mem.mar<-ir mar<-pc
              mem.data<-acc
              acc+ acc- acc<-mem acc<-acc acc<-sw)
             control-lines))
        (rlet ((pc-output (Program-counter
                             pc<-pc phi2
                             pc<-ir (low-13 ir-output)
                             pc<-pc+1 (inc pc-output)
                             pc<-sw (low-13 switches)))
               (ir-output (Instruction-reg
                             phi2 ir<-mem memory-output))
               (acc-output (Accumulator
                             acc<-acc phi2
                             acc+ (+ acc-output memory-output)
                             acc- (- acc-output memory-output)
                             acc<-mem memory-output
                             acc<-sw switches))
               (memory-output (Memory
                                phi1 phi2
                                mem.mar<-ir (low-13 ir-output)
                                mem.mar<-pc output-pc
```

```
                                        mem.data<-acc acc-output)))
                (vector acc-output
                        pc-output
                        (Vector=0? acc-output)
                        (top-3 ir-output)))))))
```

7.5.4 Specifying the Complete Computer

The complete computer can now be specified by hooking together the datapath and the controller. The output of the computer is the contents of the accumulator, the contents of the program counter, and the signal to the idle light.

```
(df (COMPUTER state ir pc acc)
  (let ((The-controller (CONTROLLER state))
        (The-datapath (DATAPATH ir pc acc)))
    (lambda (switches knob button phi1 phi2)
      (rlet (((control-lines idle-light)
              (The-controller knob button acc=0 opcode phi1))
             ((acc-output pc-output acc=0 opcode)
              (The-datapath control-lines phi1 phi2 switches)))
        (vector acc-output pc-output idle-light)))))
```

7.6 Summary

In this chapter we discussed the specification of digital systems. The broader issues of specifying digital systems was first discussed then an actual descriptive system was proposed. We proposed using an applicative langauge for specifying digital systems because it satisfied our desiderata for a design language. We showed how an applicative language is a natural way for describing complex devices which are composed of simpler devices. It is a "natural way" becuase the dataflow in functional composition directly mirrors the connections between physical devices. That is, devices specified by procedures are "wired up" by the simply composing functions. This is only possible in a functional language which supports infinite objects and delayed evaluation.

Chapter 8

Generating Net Behavior

Net behavior, introduced in Chapter 2, was employed in Chapters 4 and 5 to generate timing constraints. In this chapter we discuss some of the philosophical underpinnings of net behavior and show how it is generated. The philosophical underpinnings relate net behavior to simulation and compilation. The generation of net behavior stresses concrete algorithms and the use of logical constants in eliminating non-existent nets.

8.1 Net behavior, simulation, and compilation.

Simulators do a lot of work. Most of the work is repetitive reanalysis of the same nets over and over again. The problem is that a simulator must determine the new net partitioning that results from any change in transistor state. When the same transitions occur repeatedly (*e.g.*, every time ϕ_1 goes high), the simulator must repeatedly determine the "new" partitioning. This "new" partitioning is usually the same as some previous partitioning. Even when a high-level control strategy is used, like Mossim's Unit Delay, in which many transistors change state between partitioning, work is continually being replicated.

An analogy can be made between a circuit simulator and a language interpreter. A language has a semantics which is implemented by an interpreter, a circuit has a semantics which is implemented by a simulator. Just as an APL interpreter interprets APL programs, a simulator interprets circuits. Circuit simulators and language interpreters suffer the same problem: every time a language construct is interpreted, or transistors change state, a constant amount of the same overhead is repeatedly incurred deciding what must be done.

Compilers are used to solve the problem of repeated overhead by incorporating the decision of what to do next within the program text. Sometimes this necessitates changing the program text (called a *source level transformation*), at other times it necessitates representing the program at a level closer to the interpreter (such as translating source code into machine code the interpreter is written in). By incorporating the decision of what to do next into the program itself, the decision need only be made once at compile time.

Net behavior only enumerates a node's possible nets once. Net behavior can be considered as a compiled form of a MOS circuit. The determination of the circuit partitioning for any given state of the transistors in the circuit is done once at analysis (compile) time. A major advantage of this approach is that it is possible to analyze each possible net of a circuit fully. Steady-state voltages can be determined for each possible net (assuming reasonable assumptions about the initial values of nodes are made). The goal of research in simulators is to find new ways of avoiding the analysis of a net and still accurately predict the steady-state voltages of the net's nodes. For example, this was the major motivation behind Mossim and its switch level, order of magnitude, circuit model. Mossim repeatedly rediscovers the same nets and can't spend the time to fully analyze each newly activated net. When nets are only analyzed once, better models can be used.

8.2 Generating Net Behavior

A simple method is used to generate a circuit's behavior equations. The primitives (*i.e.*, transistors) have fixed net behavior equations. A nonprimitive circuit's net behavior equations are generated by composing the net behavior equations of the circuit's parts. This is standard practice in the field of behavioral analysis [Bobrow]. There are several advantages to this approach.

- It exploits structural hierarchy. A circuit is only analyzed once regardless of the number of times it is used.
- It is an incremental method. The work is spread over time as parts are connected instead of being done monolithically. As a result of, Silica Pithecus responses quickly.
- It localizes the use of knowledge of a circuit's logical structure. Logically disallowed nets are pruned quickly, avoiding exponential blowup. Other, non-hierarchical methods, such as path tracing, are hard to manage because of the nonlocalized (and therefore inefficient) use of logical constraints.

Net behavior equations are only created for state nodes and nodes gating transistors. They are not created for connection nodes. For example, consider the standard nMOS NAND gate (Figure 8.1). The initial voltage of node X in Figure 8.1 cannot affect the value of the output node. (Although it does affect the timing of the transitions of the output node.) Connection nodes are only partially analyzed by Silica Pithecus.

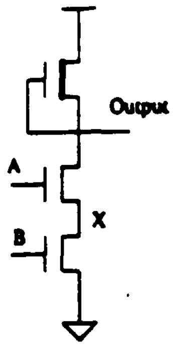


Figure 8.1: An nMOS NAND Gate. The state of internal node X does not contribute to final state of of the output the gate. X therefore does not store state and is not fully analyzed.

Silica Pithecus uses a different control strategy for generating net behavior than the one described here. The one described here is clean and easy to explain. For pragmatic reasons, the control strategy employed by Silica Pithecus is more complex. Interested readers can look at the code.

Behavior generation for a circuit C is done in three steps. The input to behavior generation is the behavior equations of C's parts. The output is the behavior equations for C. The three steps are:

1. *Renaming* the behaviors of the subcircuits of C which use names local to those subcircuits to use names of the circuit C.

2. *Merging* nets where subnodes connect. Merging is done for all nodes.

3. *Expanding* nets to span the circuit C. Expansion is the most expensive part of net behavior generation. It not performed for connection nodes.

This section has three parts corresponding to the three steps above. As a running example I will show how the behavior equations of the Three Transistor Ram (Figure 8.2) are generated. This device stores one bit on its Storage node. It is composed of three transistors, Read-Gate, Write-Gate, and Storage-Gate, and six nodes, Bus, Read, Write, Storage, A, and Gnd. The Bus node, after being connected to many other cells, will have a larger capacitance than the Storage or A nodes; Read and Write are mutually exclusive. The net behavior of this device appears in Figure 8.2. The RAM cell has a temporal logical constraint of tmutex(Read_s,Write_s). This constraint is employed during steps 2 and 3.

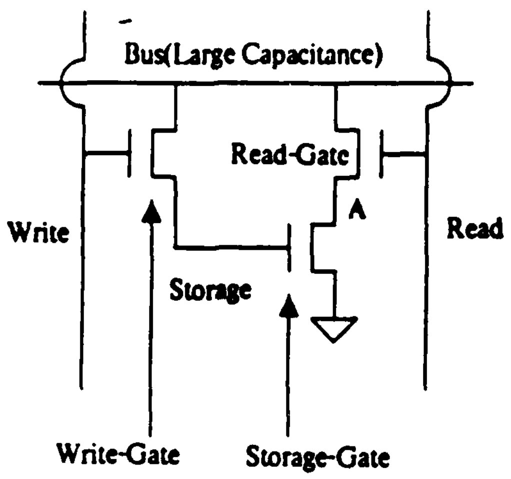


Figure 8.2: Schematic of 3 Transistor Dynamic Ram. Bus, A, Storage, Write, and Read are nodes. Read-Gate, Write-Gate, and Storage-Gate the RAM's parts. Read_b and Write_b are not simultaneously asserted.

8.2.1 Renaming Behaviors

The net behaviors of a node's subnodes[1] must be *renamed* to the structural level of the circuit whose net behavior is being generated. A net behavior equation is renamed by changing names local to the circuit's subcircuits to names used by the circuit. The following algorithm is used to effect this:

Algorithm RENAME:

Inputs: A subcircuit C, its net behavior equations E, and its renaming table R.

Outputs: New behavior equations based on E where names have been changed to refer to names used by C's parent.

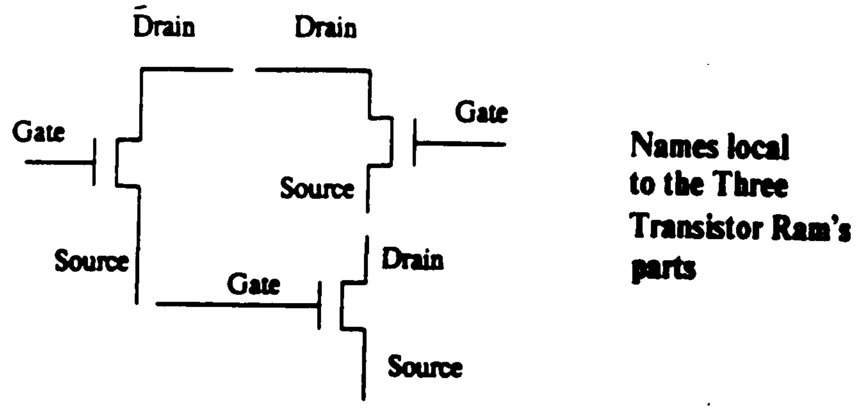
Method: Replace each name N in E with its associated name in R. ■

For example, in the Three Transistor Ram, the Source, Drain, and Gate nodes of its three transistors must be renamed to its Bus, Write, Storage, Read, A, and Gnd nodes. The relationship between the names local to the Three Transistor RAM's parts and its own local names is shown in Figure 8.3.

Consider again the behavior equations of the nMOS n-channel transistor (Table 2.1). They use names local to the transistor. Renaming the behavior of the Read-gate, for example, yields equations identical to those for the nMOS transistor except for the names used:

$\mathrm{Bus}_{net} \ = \lambda\ t\ .\ \mathrm{Read}_b(t) \Rightarrow [\mathrm{Bus}\ \mathrm{A}],$

[1]A node X's subnodes are the nodes of devices one level down the structural hierarchy which connect to make X.



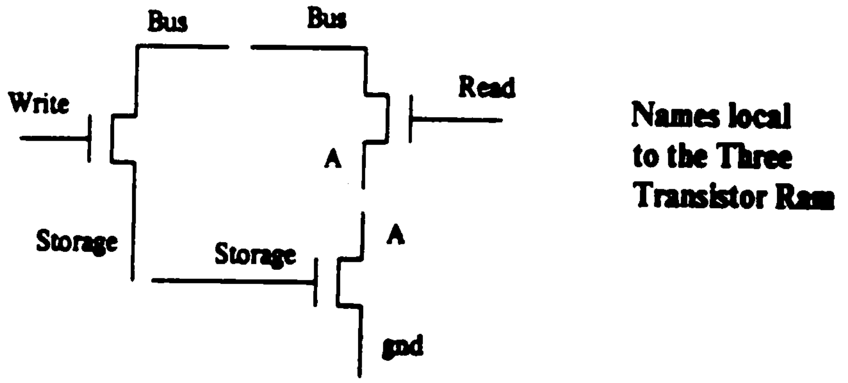


Figure 8.3: Renaming the transistors of the Three Transistor RAM.

$$
\begin{array}{lll}
 & & \texttt{NOT}(\mathrm{Read}_b(t)) \Rightarrow [\mathrm{Bus}] \\
A_{net} & = \lambda\ t\ . & \mathrm{Read}_b(t) \Rightarrow [\mathrm{Bus}\ A], \\
 & & \texttt{NOT}(\mathrm{Read}_b(t)) \Rightarrow [A] \\
\mathrm{Read}_{net} & = \lambda\ t\ . & [\mathrm{Read}]
\end{array}
$$

The two other transistors are similarly renamed.

8.2.2 Merging Nets

When subnodes connect their possible nets must be merged together. The function Merge is used to merge nets. Let N and M be nets with edges N_{edges} and M_{edges} and nodes N_{nodes} and M_{nodes}, respectively. Then (Merge N M) = O where $O_{edges} = N_{edges} \cup M_{edges}$ and $O_{nodes} = N_{nodes} \cup M_{nodes}$. Merge can be extended to expect an arbitrary number of arguments.

Behavior expressions are merged together using the following algorithm:

Algorithm MERGE NET EXPRESSIONS:

Inputs: A node and the net behavior expressions of its subnodes. There is one expression per subnode.

Outputs: An approximation to the net behavior expression of the node.

Method: Apply the function Merge to the node's subnodes' net expressions. Then rewrite the resulting expression into *standard form*. An expression is in standard form when 1) the top level expression is a conditional, 2) all its predicates consist of node names and boolean operators, and 3) all its consequents are nets. Once in standard form, the temporal logical constraints of the circuit are used to prune *logically disallowed clauses*. A clause is logically disallowed when its predicate is always false. ■

For example, two subnodes connect to make the Bus node of the Three Transistor Ram. Two net behavior equations (one for each subnode) exist for the Bus node:

$$\begin{aligned}\text{Bus}_{net} = \lambda\ t\ .\ & \text{Read}_b(t) \Rightarrow [\text{Bus A}],\\ & \texttt{NOT}(\text{read}_b(t)) \Rightarrow [\text{Bus}].\end{aligned}$$

and

$$\begin{aligned}\text{Bus}_{net} = \lambda\ t\ .\ & \text{write}_b(t) \Rightarrow [\text{Bus Storage}],\\ & \texttt{NOT}(\text{write}_b(t)) \Rightarrow [\text{Bus}].\end{aligned}$$

Merge is applied to the expressions to yield an approximation of the behavior equation for the bus node:

$$\begin{aligned}\text{bus}_{net} = \texttt{Merge}(\lambda\ t\ .\ & \text{Read}_b(t) \Rightarrow [\text{Bus A}],\\ & \texttt{NOT}(\text{Read}_b(t)) \Rightarrow [\text{Bus}].,\\ \lambda\ t\ .\ & \text{Write}_b(t) \Rightarrow [\text{Bus Storage}],\\ & \texttt{NOT}(\text{Write}_b(t)) \Rightarrow [\text{Bus}].)\end{aligned}$$

This equation is rewritten into standard form by forming the cross product of the clauses of the conditionals:

$$\begin{aligned}\text{Bus}_{net} = \lambda\ t\ .\ & \text{Read}_b(t) \wedge \text{Write}_b(t) \Rightarrow \texttt{Merge}([\text{Bus A}], [\text{Bus Storage}]),\\ & \text{Read}_b(t) \wedge \texttt{NOT}(\text{Write}_b(t)) \Rightarrow \texttt{Merge}([\text{Bus A}], [\text{Bus}]),\\ & \texttt{NOT}(\text{Read}_b(t)) \wedge \text{Write}_b(t) \Rightarrow \texttt{Merge}([\text{Bus}], [\text{Bus Storage}]),\\ & \texttt{NOT}(\text{Read}_b(t)) \wedge \texttt{NOT}(\text{write}_b(t)) \Rightarrow \texttt{Merge}([\text{Bus}], [\text{Bus}])\end{aligned}$$

After executing the Merge's we get

$$\begin{aligned}\text{Bus}_{net} = \lambda\, t\,.\ & \text{Read}_b(t) \wedge \text{Write}_b(t) \Rightarrow [\text{Bus A Storage}],\\ & \text{Read}_b(t) \wedge \texttt{NOT}(\text{Write}_b(t)) \Rightarrow [\text{Bus A}],\\ & \texttt{NOT}(\text{Read}_b(t)) \wedge \text{Write}_b(t) \Rightarrow [\text{Bus Storage}],\\ & \texttt{NOT}(\text{Read}_b(t)) \wedge \texttt{NOT}(\text{Write}_b(t)) \Rightarrow [\text{Bus}]\end{aligned}$$

A simplifier now removes logically disallowed clauses. Because Read$_b$ and Write$_b$ are never simultaneously asserted, the first clause is logically disallowed and is removed. Note that a clause can be logically disallowed not only because of mutually exclusive signals, but also for any fixed relationship between signals. For example, if it were the case that either Read$_b$ or Write$_b$ were always asserted, then the last clause would be logically disallowed.

Besides eliminating logically disallowed clauses, the simplifier also simplifies predicates to shrink them. For example, because Read$_b$ and Write$_b$ are mutually exclusive, Read$_b$ implies NOT(Write$_b$) and vice versa. The result of this simplification is

$$\begin{aligned}\text{Bus}_{net} = \lambda\, t\,.\ & \text{Read}_b(t) \Rightarrow [\text{Bus A}],\\ & \text{Write}_b(t) \Rightarrow [\text{Bus Storage}],\\ & \texttt{NOT}(\text{Read}_b(t)) \wedge \texttt{NOT}(\text{Write}_b(t)) \Rightarrow [\text{Bus}].\end{aligned}$$

Figure 8.1 shows the result of applying Algorithm MERGE NET EXPRESSIONS to all the nodes of the Three Transistor RAM.

8.2.3 Expanding Nets

Merging only solves half the problem of generating equations. Nets which span across many devices (*e.g.*, from the ground node to the Bus node of the Three Transistor RAM) must be found. The generation of behavior equations is completed by iteratively applying a simple expansion transformation to each node. The expansion transformation expands the "boundaries" of the possible nets of the node to include adjacent nets.

The expansion transformation replaces a chosen node Y in a net N by the expression for Y_{net}. This extends N to include the possible nets of Y. The replacement can result in the original net N being replaced by multiple nets. The replacement is very similar to the merging of nodes discussed above.

Algorithm EXPAND NET BEHAVIOR EQUATIONS:

Inputs: The approximate net behavior equations for a circuit *C* that were created by Algorithm Merge.

Outputs: The complete net behavior expressions for *C*'s nodes.

Method:

$$\begin{array}{lll}
\text{Bus}_{net} & = \lambda\ t\ . & \text{Read}_b(t) \Rightarrow [\text{Bus A}], \\
 & & \text{Write}_b(t) \Rightarrow [\text{Bus Storage}], \\
 & & \texttt{NOT}(\text{Read}_b(t)) \land \texttt{NOT}(\text{write}) \Rightarrow [\text{Bus}] \\
\text{Storage}_{net} & = \lambda\ t\ . & \text{Write}_b(t) \Rightarrow [\text{Bus Storage}], \\
 & & \texttt{NOT}(\text{Write}_b(t)) \Rightarrow [\text{Storage}] \\
\text{A}_{net} & = \lambda\ t\ . & \text{Read}_b(t) \land \text{storage}_b(t) \Rightarrow [\text{Gnd A Bus}], \\
 & & \text{Read}_b(t) \land \texttt{NOT}(\text{storage}_b(t)) \Rightarrow [\text{A Bus}], \\
 & & \texttt{NOT}(\text{read}_b(t)) \land \text{storage}_b(t) \Rightarrow [\text{Gnd A}], \\
 & & \texttt{NOT}(\text{read}_b(t)) \land \texttt{NOT}(\text{storage}_b(t)) \Rightarrow [\text{A}] \\
\text{Read}_{net} & = \lambda\ t\ . & [\text{read}] \\
\text{Write}_{net} & = \lambda\ t\ . & [\text{write}]
\end{array}$$

Table 8.1: Equations describing the net behavior of Three Transistor RAM after merging. These are not the final equations. Not all possible nets have been generated.

```
For the net behavior expression EX of each equation do
  Repeat
    Choose some net N of EX
    Choose some node Y in the N that has not been chosen before.
    Replace N by (Merge N $Y_{net}$).
    Rewrite EX back to standard form.
    Eliminate disallowed clauses.
Until closure is reached (i.e., nets stop expanding). ■
```

For example, consider Bus_{net} of Table 8.1. Choosing to expand node A in this expression results in:

$$\begin{array}{lll}
\text{Bus}_{net} = \lambda\ t\ . & \text{Read}_b(t) \land \text{storage}_b(t) \Rightarrow [\text{Bus A Gnd}], \\
 & \text{Read}_b(t) \land \texttt{NOT}(\text{Storage}_b(t)) \Rightarrow [\text{Bus A}], \\
 & \text{Write}_b(t) \Rightarrow [\text{Bus Storage}], \\
 & \texttt{NOT}(\text{Read}_b(t)) \land \texttt{NOT}(\text{Write}_b(t)) \Rightarrow [\text{Bus}]
\end{array}$$

Applying the transformation again for any other nodes in the possible nets of Bus results in no change, so this is the final, and correct, behavior expression for the Bus node. Repeating the expansion for the other nodes yields the final set of behavior expressions for the Three Transistor RAM Table 8.2. Because the A node is a connection node its behavior equation is not computed.

$$\begin{aligned}
\mathrm{Bus}_{net} &= \lambda\, t\, .\ \mathrm{Read}_b(t) \wedge \mathrm{Storage}_b(t) \Rightarrow [\mathrm{Bus\ A\ Gnd}],\\
&\qquad \mathrm{Read}_b(t) \wedge \mathtt{NOT}(\mathrm{Storage}_b(t)) \Rightarrow [\mathrm{Bus\ A}],\\
&\qquad \mathrm{Write}_b(t) \Rightarrow [\mathrm{Bus\ Storage}],\\
&\qquad \mathtt{NOT}(\mathrm{Read}_b(t)) \wedge \mathtt{NOT}(\mathrm{Write}_b(t)) \Rightarrow [\mathrm{Bus}]\\
\mathrm{Storage}_{net} &= \lambda\, t\, .\ \mathrm{Write}_b(t) \Rightarrow [\mathrm{Bus\ Storage}],\\
&\qquad \mathtt{NOT}(\mathrm{Write}_b(t)) \Rightarrow [\mathrm{Storage}]\\
\mathrm{Read}_{net} &= \lambda\, t\, .\ [\mathrm{Read}]\\
\mathrm{Write}_{net} &= \lambda\, t\, .\ [\mathrm{Write}]
\end{aligned}$$

Table 8.2: Equations describing the behavior of Three Transistor RAM. The A node is not represented because, as a connection node, its behavior expression is not computed.

(As described net expansion is a very costly operation because the same nets will be merged many times. Silica Pithecus carefully keeps track of which mergings need to be done to minimize the number of mergings.)

Chapter 9

Net Behavior to Digital Behavior

As explained in Chapter 6, Silica Pithecus abstracts a circuit's digital behavior directly from the circuit's net behavior. The translation from net behavior to signal behavior and then signal behavior to digital behavior only occurs subliminally. Nonetheless, we will act as if the translation did occur: doing so simplifies the exposition.

In the examples the outputs of a circuit's signal behavior applied to the circuit's inputs were abstracted. In this chapter (a representation of) the signal behavior itself is abstracted. Applying this abstraction to some inputs $\hat{\imath}$ yields the same result as abstracting the outputs of the original signal behavior applied to $\hat{\imath}$. That is, whereas before we were computing $\lambda\ \hat{\imath}\ .\ \mathrm{ABS}(B_S(\mathit{Design})(\hat{\imath}))$, we now compute $\mathrm{ABS}(B_S(\mathit{Design}))(\hat{\imath})$. In the second expression $\mathrm{ABS}(B_S(\mathit{Design}))$ is applied to $\hat{\imath}$. We have made ABS polymorphic; here ABS is operating on functions, rather than on signals. Signal behavior itself is abstracted by constant folding the (representation of the) signal abstraction function, $\mathrm{ABS}_{S \Rightarrow T}$, over the (representation of the) circuit's signal behavior. This chapter shows how $\mathrm{ABS}(B_S(\mathit{Design}))$ is derived from *Design*'s net behavior.

The formal notation for describing the abstracted signal behavior of a circuit with N inputs (which are signals) and M outputs (which are digital values) is

$$\lambda\ \hat{s}\ .\ \hat{o}\ \textbf{where}\ o_1 = f_1(\hat{s},\hat{o}), \ldots, o_m = f_m(\hat{s},\hat{o}).$$

The formal notation is rarely used. Instead, the equational part will be given; the lambda bound signals and return values will be left implicit. An example appears in Table 9.1, which shows the abstracted behavior of the Inverting Latch.

Conceptually, the generation of $\mathrm{ABS}(B_S(\mathit{Design}))$ is a three step process:

1. Constraints are generated from *Design*'s net behavior,
2. $B_S(\mathit{Design})$ is generated,
3. $B_S(\mathit{Design})$ is abstracted to yield $\mathrm{ABS}(B_S(\mathit{Design}))$.

During the first step flow and timing constraints are generated. During the third step threshold constraints are generated, and charge sharing bugs and ratio bugs are found and reported. In the implementation the steps 2 and 3 are intertwined to minimize the amount of detail of $B_S(Design)$ which is generated.

The digital value at node X is denoted X_{bf}. We retract the earlier statement that the digital values are 0, 1, and *float*.[1] Instead, four digital values, called driven 1, driven 0, floating 1, and floating 0, are used. Digital values have two attributes, a boolean attribute and a floating attribute. Each attribute has two possible values, which gives rise to the four digital values. A digital value X_{bf} is denoted $<X_b,X_f>$ where X_b denotes the boolean attribute of X and X_f denotes the floating attribute of X. The four digital values are denoted <1,D>, <0,D>, <1,F>, and <0,F>.

Two functions, B and F, are used to project the different parts of digital values. B projects the boolean part, and F projects the floating part.

The greater precision of the four value scheme is required. The three value scheme (0, 1, and *float*), though adequate for describing intended digital behavior, is inadequate as a target space for abstraction. The problem is that a signal may either be used for its boolean attribute or for its floating attribute. The abstraction function can't know which attribute is desired by the designer and must return both attributes. The comparator is responsible for matching abstracted attributes with the designer's intentions. A 1 intended by the designer is matched by either <1,D> or <1,F>. Similarly, a *float* intended by the designer is matched by either <0,F> or <1,F>.

Threshold constraints dictate that certain signals cannot suffer threshold drops. A signal that has suffered a threshold drop is called *degraded*. Threshold constraints ensure that ratios of nMOS gates will yield logically valid voltages. They also ensure that voltage levels are aren't made invalid by passing through pass transistors gated by degraded signals. The example of Chapter 3 indicated that threshold constraints were generated after $B_S(Design)$ was generated. This is not the case. Rather, threshold constraints are generated as $B_S(Design)$ is generated.

Example: Inverting Latch

As a running example, this chapter shows how ABS(B_S(*Inverting Latch*)) (Table 9.1) is derived from the Inverting Latch's net behavior. The net behavior of the Inverting Latch (Figure 1.3) is

$$\begin{aligned}\mathrm{Out}_{net} &= \lambda\, t\,.\ S_b(t) \rightarrow [\mathrm{Vdd, Gnd, Out}],\\ &\qquad \mathtt{NOT}(S_b(t)) \rightarrow [\mathrm{Vdd, Out}]\\ \mathrm{In}_{net} &= \lambda\, t\,.\ \mathrm{Latch}_b(t) \rightarrow [\mathrm{In, S}],\\ &\qquad \mathtt{NOT}(\mathrm{Latch}_b(t)) \rightarrow [\mathrm{In}]\end{aligned}$$

[1]The digital specification language still uses just the three values. But Silica Pithecus's internal representations use the four value scheme.

$$\begin{aligned}
\text{Out}_{bf} =\ & \text{B}(\text{S}_{bf}) \rightarrow <0,\text{D}> \\
& \neg\text{B}(\text{S}_{bf})) \rightarrow <1,\text{D}>. \\
\text{In}_{bf} =\ & \text{ABS}(\text{In}_{s}) \\
\text{S}_{bf} =\ & \text{B}(\text{Latch}_{bf}) \rightarrow \text{ABS}(\text{In}_{s}), \\
& \neg\text{B}(\text{Latch}_{bf}) \rightarrow \text{ABS}(\text{S}_{s}). \\
\text{Latch}_{bf} =\ & \text{ABS}(\text{Latch}_{s})
\end{aligned}$$

Table 9.1: ABS(B_S(*Inverting Latch*)). X_{bf} denotes the digital behavior of X.

$$\begin{aligned}
\text{S}_{net} &= \lambda\ t\ .\ \text{Latch}_{b}(t) \rightarrow [\text{In, S}], \\
&\qquad \texttt{NOT}(\text{Latch}_{b}(t)) \rightarrow [\text{S}] \\
\text{Latch}_{net} &= \lambda\ t\ .\ [\text{Latch}].
\end{aligned}$$

The contraints issued while generating the above digital behavior are:

1. `falls`(Latch$_s$) $\Rightarrow$ `falls-first`(Latch$_s$,In$_s$)

2. `control`(Latch$_s$)

3. Latch$_b$ $\Rightarrow$ `overpowers`($[\text{In}]_{In}$, $[\text{S}]_{S}$)

4. `no-drop`(Latch$_s$)

(The generation of the first three constraints has been discussed (Chapters 4 and 5). This chapter discusses generating the fourth constraint.)

Creating ABS(B_S(*Design*)) and generating theshold constraints is conceptually simple. Starting from net behavior, each transistor in a net is replaced by its full model. If there are N transistors in a net this generates 3^N *detailed nets*. A detailed net is a net where each transistor has been replaced by the resistor and threshold drop device which model one of the transistor's operating regions (Figure 9.1 shows a detailed net). Each detailed net is then analysed to produce a combination of threshold contraints, abstracted voltage levels, and reports of actual design errors. Although 3^N detailed nets are theoretically required, only about $2N$ detailed nets actually need to be enumerated.

This chapter has two sections. The first section shows the analyis of detailed nets. The second section presents the enumeration method for detailed nets which minimizes the number of detailed nets generated.

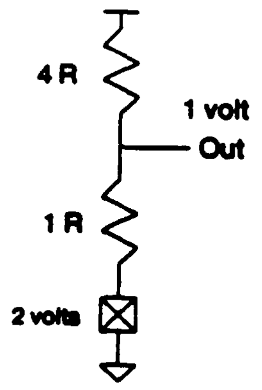


Figure 9.1: A Detailed Net

9.1 Analyzing Detailed Nets

During analysis each net spawns many detailed nets. The detailed nets are analysed to predict and abstract the voltage at a given node in the detailed net. When valid voltages occur they are abstracted to either 0 or 1. When an invalid voltage occurs either a threshold constraint is generated, a charge sharing bug is reported, or a threshold bug is reported.[2]

A signal value is the value of a signal at some particular time. A signal value has two attributes, strength and voltage. Once all constraints have been issued and design errors caught, all signals are abstractable. Therefore no errors are reported when abstracting a signal and all that is left for the abstraction function to do is turn a signal's final value into a digital value. The following function does this.

Algorithm STD (Signal Value to Digital Value):
Inputs: A signal value SV
Outputs: A digital Value.
Method:

```
If SV is driven let strength be "D" else let it be "F."
  If V > 3.5 then return <1,strength>
  else if V < 1.5 then return <0,strength>.
```

We now discuss determining and abstracting the voltage at a given node in a detailed net. When the final voltage of a signal S can't be determined (because

[2]This is overly conservative. If the designer depends only on the signal being undriven and not on the signal's boolean value, then no error should be generated. Silica Pithecus could be changed to return a value denoting an invalid voltage. Then only when the designer depended on the value would an error be generated.

the voltage comes from stored charge rather than from gates), it is known, because of constraints and assumptions of valid inputs, that the final voltage will be valid. Abstracting such a signal yields the symbolic result ABS(S).

There are three cases to consider depending on the nodes in the detailed net *Net*:

***Net* contains driven nodes.** The driven nodes determine the final values of all the undriven nodes in the net. With one exception, nets may contain only one driven node. The exception is that a net may contain both Vdd and Gnd. When a net contains both Vdd and Gnd, the network is transformed into an equivalent voltage divider using Thevenin equivalents. (Examples appear below.)

***Net* contains no driven or input nodes.** There are two cases to consider based on whether there is some node N whose capacitance overpowers the rest of the net. When N exists, the result of abstraction is the symbolic result ABS(N_s).

When N does not exist, data dependent methods must be used to determine if charge sharing will result in valid logic levels. These methods show that enough nodes have the same logical value such that the net settles to this logical value. Silica Pithecus employs a simple heuristic to determine whether enough nodes will have the same logical value. It first determines the phase at which the net is activated (*e.g.*, ϕ_1 or ϕ_2).[3] It then determines the values of the nodes at the end of the previous phase. If enough of the nodes have the same value then that value is used. When not enough are the same, an error is generated.

***Net* contains an input node I.** Flow constraints guarantee that I is the dominant node. The voltage of a node other than I in *net* cannot be determined, but the abstraction of a signal anywhere in *net* is guaranteed to be the same as ABS(I_s). (Note that the term "input node" refers to nodes declared to be the inputs of circuits. "Input node" does not refer to driven nodes, as it does in other systems. There is no guarantee that an input node will be connected to a driven node.)

9.2 Enumerating Detailed Nets

A conducting transistor has three regions: full conductance (gate voltage of 5 volts), partial conductance (gate voltage above 3.5 volts), and unknown conductance (gate voltage between 1.5 and 3.5 volts). Theoretically, to predict the full behavior of

[3]On the next revision of this document I will explain where ϕ_1 and ϕ_2 came from.

a given net, the net must be instantiated with all ranges of its transistors' behaviors. Therefore predicting the behavior of a net theoretically requires generating 3^N detailed nets where N is the number of transistors in the net.

However, by carefully enumerating the nets and by being conservative,[4] only $2N$ detailed nets need be generated. There are two key observations. First, there is no need to create detailed nets where a transistor is in the unknown conductance region. By implicitly constraining all signals which gate transistors to have valid voltages it is ensured that transistors never operate in this region. These implicit constraints are checked as part of verification. This strategy reduces the theoretical number of required detailed nets to 2^N.

Second, by judiciously choosing the order in which nets are enumerated, it is possible to set the operating region of transistor to just one of the two remaining regions it could be in. The region is set the first time the transistor is encountered. When a detailed net yields a valid voltage where transistor T is partially conducting, then the net will yield a valid voltage when T is fully conducting. Therefore there is no need to generate the detailed net where T is fully conducting. It is only necessary to generate that net when the detailed net yields invalid voltages when T is partially conducting. In this case, a threshold constraint is issued to prevent T from partially conducting and thenceforth T is always fully conducting. Therefore detailed nets where T is partially conducting need never be generated. These two strategies together, plus the rule that the possible nets of a node are analysed from smallest to largest, allow the analysis routine to generate only a linear number of detailed nets.

[I must mention validifying use of foo_b and must mention steadystate logical constraints here.]

Example: Inverting Latch

As an example we will generate ABS(B_S(*Inverting Latch*)) from Inverting Latch's net behavior (Figure 9.2). We will analyze the behavior of each of its nodes in the order Latch, In, S, and Out. The result appears in Table 9.1.

The first node to be analyzed is Latch. It has only one possible net, itself. Therefore, the abstraction of Latch's signal behavior is the symbolic result ABS(Latch_s).

The second node to be analyzed is In. It has two possible nets. But because it is an input node, generated constraints guaranteed that In is the dominant node of all of its possible nets. Therefore the abstraction of its signal behavior is ABS(In_s).

The third node to be analysed is S. It has two possible nets. They must be analysed in detail. The smaller net is analysed first. This net is selected when voltage($\text{Latch}_s(t_f)$) < 1.5 volts. Because the net has just one node S, the abstrac-

[4]Conservatively means, as always, at the risk of introducing a few more sources of false negatives into the verifier.

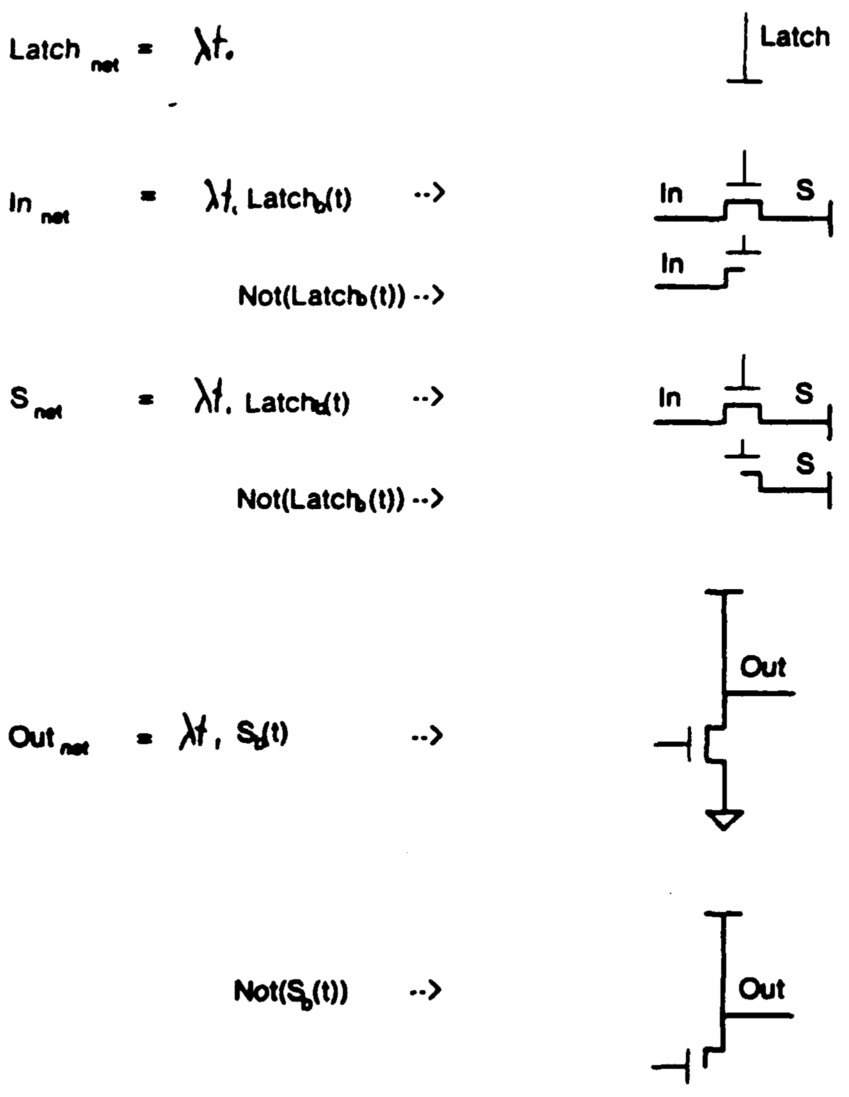

$\text{Latch}_{net} = \lambda t.$
Latch
$\text{In}_{net} = \lambda t.\ \text{Latch}_b(t)$ -->
In
S
Not(Latch$_b$(t)) -->
In
$\text{S}_{net} = \lambda t.\ \text{Latch}_d(t)$ -->
In
S
Not(Latch$_b$(t)) -->
S
$\text{Out}_{net} = \lambda t.\ \text{S}_d(t)$ -->
Out
Not(S$_b$(t)) -->
Out

Figure 9.2: Inverting Latch's Net Behavior

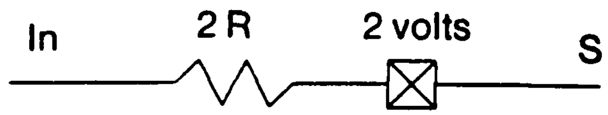
tion of the net's signal at node S is ABS(S_s). The first of the two clauses of S's abstracted behavior expression is now generated. It is

$$\text{voltage}(\text{Latch}_s(t_f)) < 1.5 \Rightarrow \text{ABS}(\text{S}_s).$$

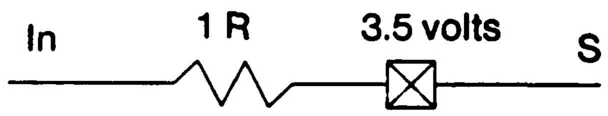
The predicate is equivalent to, and replaced by, $\neg$B(Latch$_{b,f}$) to yield

$$\neg\text{B}(\text{Latch}_{b,f}) \Rightarrow \text{ABS}(\text{S}_s).$$

S's other possible net is then analysed. The first detailed net generated uses a partially conducting Latch-Gate:



Analyzing this detailed network yields a maximum voltage of 2 volts at S due to threshold drops. This is an invalid voltage, so a threshold constraint is slapped on Latch$_s$. (This is the fourth, and final, constraint placed on the Inverting Latch.) In all future detailed nets this constraint will guarantee Latch-Gate is always fully conducting. The detailed net where Latch-Gate is fully conducting is tried next:



This net has no threshold problems. We employ the flow constraint that requires $[\text{In}]_{In}$ overpowers $[\text{S}]_S$ to infer that the abstraction of the signal at S is ABS(In$_s$). Therefore, the second clause of S's abstracted behavior expression is

$$\text{voltage}(\text{Latch}_s(t_f)) = 5 \Rightarrow \text{ABS}(\text{In}_s)$$

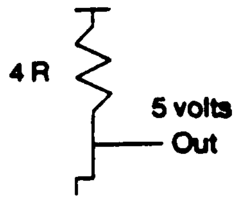
which is equivalent to is

$$\text{B}(\text{Latch}_{b,f}) \Rightarrow \text{ABS}(\text{In}_s).$$

We therefore have

$$\begin{aligned} \text{S}_{b,f} = {} & \text{B}(\text{Latch}_{b,f}) \rightarrow \text{ABS}(\text{S}_s), \\ & \neg\text{B}(\text{Latch}_{b,f}) \rightarrow \text{ABS}(\text{In}_s). \end{aligned}$$

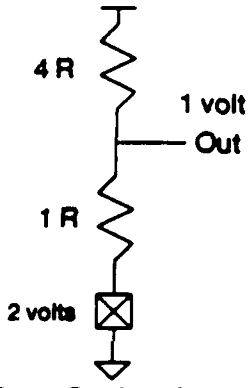
The last node to be analysed is Out. It has two possible nets. The detailed net for the smaller net is



Analysis yields 5 volts at Out. One clause of Out's abstracted signal behavior is therefore

$$B(S_{b,f}) \Rightarrow <1,D>.$$

The first detailed net for Out's other possible net is



Analysis yields 1 volt at Out. Out's other abstracted clause is

$$B(S_{b,f}) \Rightarrow <0,D>.$$

There is no need to generate the other detailed net as it will yield a valid voltage when the pulldown is partially conducting.

We therefore have

$$\begin{aligned} Out_{b,f} = \; & B(S_{b,f}) \rightarrow <0,D> \\ & \neg B(S_{b,f}) \rightarrow <1,D>. \end{aligned}$$

9.3 Summary

This chapter showed how ABS(B_S(*Design*)) was generated from *Design*'s net behavior. Silica Pithecus's method for determining the final voltages of nodes in detailed nets was given. When invalid voltages are detected Silica Pithecus issues an appropiate threshold constraint, charge sharing bug report, or invalid ratio bug report. Silica Pithecus generates detailed nets from a node's possible nets. Although exponentially many detailed nets can be generated for each possible net, judicious engineering allows Silica Pithecus to generate just a linear (in the number of transistors) number of detailed nets for any given possible net.

Chapter 10

Hierarchical Verification

I claim that despite the ubiquity of the word "hierarchical" in many systems and theses, researchers have missed the major idea of hierarchical verification. A verifier is a theorem prover, and hierarchical verification is a mechanism for focusing the theorem prover. An effect of focusing the theorem prover is vastly increased efficiency. Researchers have emphasized the efficiency gains without understanding the global picture.

This chapter has 5 sections. The first section presents hierarchical verification as a method for focusing for proofs. The second second discusses Barrow's hierarchical verifier Verify. The third section presents the interactions between hierarchical verification and multilevel verification. Hierarchical digital verification is presented in the fourth section. Problems with hierarchical functional verification are discussed in the fifth section.

10.1 Focusing Proofs

I claim that verification is formally proving that certain predicates about a circuit hold. A verifier is a theorem prover which proves the predicates are true (or proves they are false). The predicates can be about physical characteristics of the circuit (*e.g.*, design rules), about electrical properties (*e.g.*, all outputs have valid voltages), or functional properties (*e.g.*, the circuit is a 32 bit adder). In each of these cases, when a design is represented solely in terms of the interconnections of its primitives without any imposed structure, a theorem prover must wade through a massive amount of information to derive a proof. The theorem prover will require a long time to derive the proof, if it can find one at all.

Large, complex designs are created by recursively composing simpler designs to create more complex designs. The structure that results can be, and has been, exploited to organize a correctness proof. A hierarchical verifier is a method of verification that establishes lemmas (subpredicates) about the design, then proves

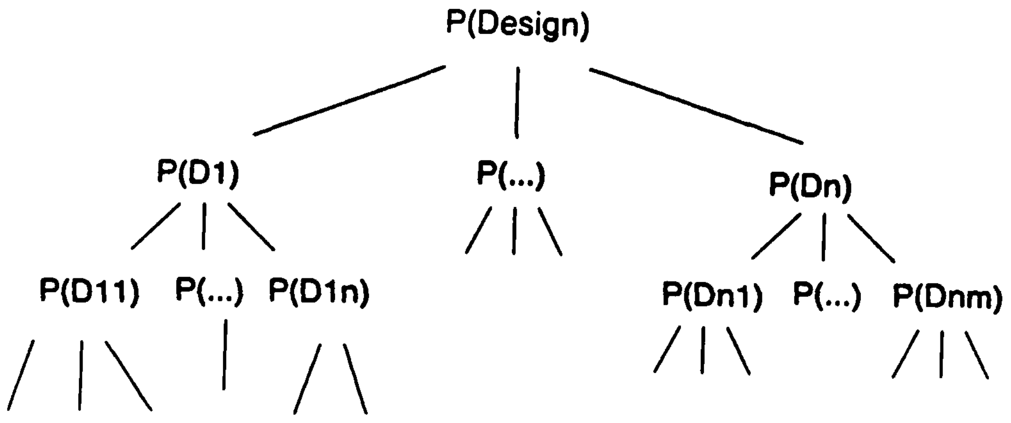


Figure 10.1: Focusing a Proof. When described hierarchically, a design can be represented as tree. The leaves of the tree represent the primitive elements of the design and the nodes represent composition. Shared structure occurs when a node is a child of more than one other node. In hierarchical verification, lemmas are established at nodes of the tree. These lemmas are used to establish the proof either the proof itself or other lemmas higher in the tree.

---

the desired predicate from the lemmas. The lemmas are organized around the structural components of a design. For some types of verifiers, such as design rule checkers, the lemmas can be automatically found (*e.g.*, the inside of this cell is OK). For other verifiers, such as Verify or Silica Pithecus which are behavioral verifiers, the lemmas must be provided by the designer.

Hierarchical verification is shown pictorially in Figure 10.1. A hierarchical design can be represented as a tree where the leaf nodes represent primitives (*e.g.*, rectangles, transistors, switches, adders) and the nodes represent compositions (*e.g.*, abutment, functional composition) of components. Structure can be shared, a node can be a child of many other nodes. Whereas a non-hierarchical verifier (hereafter called a *flat verifier*) would reason directly from the primitives and the meaning of compositions, a hierarchical verifier proves lemmas about nodes in the tree. These lemmas are used to prove lemmas further up in the tree. Finally, the lemmas are used to verify the design itself.

For example, when asked to verify an ALU, a hierarchical verifier would first verify that the low level cells, such as adder cells and comparator cells are correct, then it would prove that arrays of those cells are correct (using the knowledge that the cells are correct), then it would prove the ALU itself is correct (using the knowledge that the arrays of cells are correct).

There are three major advantages of hierarchical verification. First, the intermediate lemmas may allow a proof to be found where a flat verifier fails to find a proof. Second, a component's lemmas need only be established once, regardless of the number of times the component is used. Third, establishing lemmas causes errors to be pinpointed quickly and accurately.

In regular designs a given component may be used many times. When shared structure is used to implement the different instantiations of the component, lemmas need only be established once for it (at the node in the tree which creates the component). A flat verifier is forced to implicitly reestablish the lemmas for each occurrence of of the component. This advantage, only establishing lemmas once, is cited by other researchers as the primary contribution of hierarchical verification. The gain of establishing lemmas only once for a component depends on the regularity of the design. For extremely regular designs with components replicated many times there is a large gain, for less regular designs there is less gain.

Establishing lemmas for components is equivalent to verifying those components. Verifying components before verifying the whole design finds errors incrementally, which is extremely important. Detecting bugs as they occur causes reevaluation of designs at an early stage. Often a design flaw in a small segment of a design invalidates the rest of the design. Detecting the flaw only after the design is completed is wasteful.

10.2 Behavioral Verifiers: Verify

To make these ideas concrete without introducing the complexity of dealing with a design's context, we will discuss how Verify [Barrow], a hierarchical verifier, works. Verify is a monolevel verifier that operates in the digital and arithmetics domains.

Verify has two inputs, a structural description of a digital design, and a functional description (*i.e.*, a program) of the intended behavior of the design. Verify derives the behavior exhibited by the structural description by functionally composing the behaviors of the structure's parts. It then compares the derived and intended behaviors for equivalence. If the behaviors are equivalent, the structural description is declared to be a valid implementation of the intended behavior. The lemmas that Verify proves at each node is that the behavior of each component exhibits its specified behavior.

Verify derives its power from its method for deriving a structure's behavior. It composes the intended behaviors (the lemmas) of a structure's parts rather than the derived behavior of the parts. The resulting behavioral description is far easier to manipulate and reason about than the behavioral description that would have been otherwise generated. For example, once an n-bit adder is verified as an adder, its behavior is the arithmetic sum of its inputs, rather than a complex boolean formula of n-bit boolean vectors.

10.3 Hierarchical Multilevel Verifiers: The Problem of Context

(When discussing hierarchical multilevel verifiers, there is a potential confusion to be avoided. There are two hierarchies, the *descriptive hierarchy* which refers to different levels of behavioral description, and the *structural hierarchy* which refers to how a design is put together. When just the word "hierarchy" is used the intended hierarchy should be clear from context.)

Monolevel verifiers assume a design's behavior to be solely a function of the interconnection of the design's parts, and not a function of the context of the design. This assumption cannot be made for multilevel verifiers. Employing partial abstraction functions and specialized behaviors requires considering a design's context when determining its behavior: because a designer only cares about a subset of the possible behaviors of a design, the context that gives rise to that subset is important.

Nonetheless, just as hierarchical monolevel verifiers use the intended behavior of components when proving a circuit exhibits a given behavior, we would like to the same when performing multilevel verification. Doing so is much more efficient than abstracting the combination of the circuit's signal behaviors. How hierarchical multilevel verifiers operate will be described both graphically and algebraically.

The relationship between multilevel verification and hierarchical multilevel verification is shown graphically in Figure 10.2. In this figure the design's concrete behavior is on the left and its intended abstract behavior is on the right. The structural representation is on the top and the behavioral representation on the bottom. The task is to verify the top left corner (the structurally represented design where the concrete behaviors of the components are known) against the lower right hand corner (the intended abstract behavior of the whole design). There are two ways to perform this task, corresponding to paths $A \rightarrow B \rightarrow C$ (flat verification) and $A \rightarrow D \rightarrow C$ (hierarchical verification). Flat verification first derives the concrete behavior of the whole from its parts, then abstracts this behavior and compares it to the intended abstract behavior. Hierarchical verification first verifies the components, then composes the components' intended abstract behaviors to derive the abstracted behavior of the whole design. This derived description is then compared against the intended abstract behavior of the entire design.

This can be stated algebraically using the function *compose*, which forms the composition of its operands.[1] Flat verification shows that

$$\mathrm{ABS}(compose_{con}(B_{con}(D_1), \ldots, B_{con}(D_n))(\hat{\imath}))$$

[1] This treatment is not formal, for a formal treatment of the *compose* operator see [Gordon81]. (He calls this operator "|.")

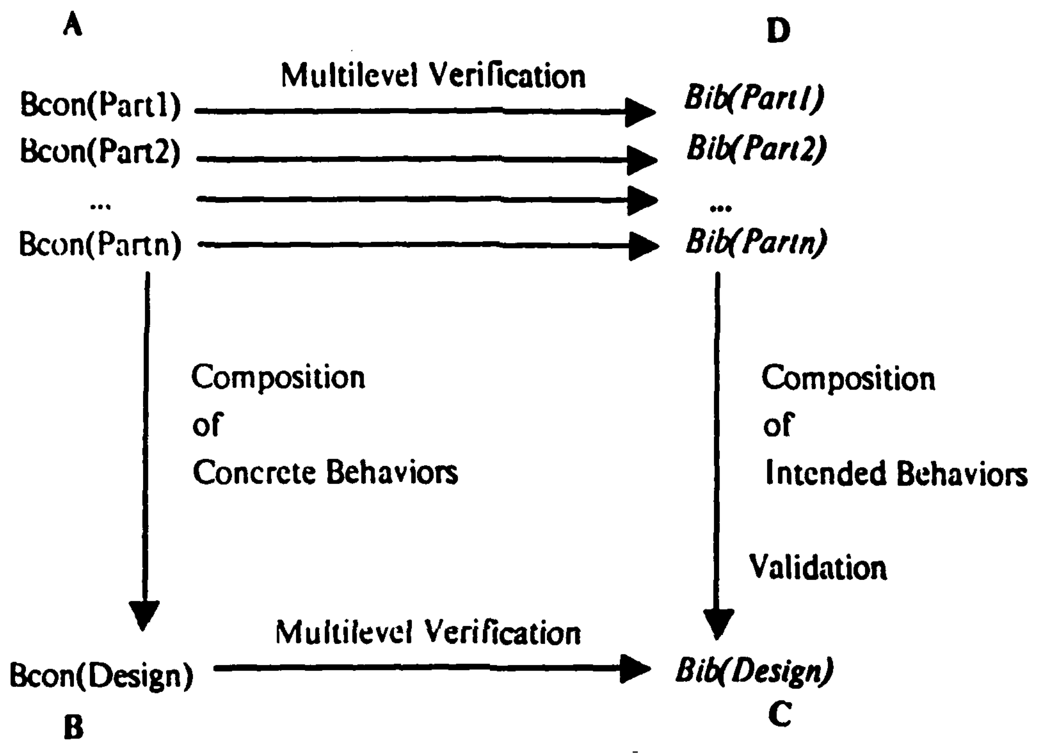


Figure 10.2: The relationship between multilevel verification and hierarchical multilevel verification. In multilevel verification (path A → B → C) the concrete behaviors of the components are composed, the resulting behavior abstracted, then the abstracted behavior is validated against the intended behavior. In hierarchical multilevel verification (path A → D → C) the structure's parts are verified, the the parts' intended abstract behaviors are composed, and the resulting behavior validated against the intended abstract behavior of the whole.

$$=$$

$$B_{IB}(Design)(\text{ABS}(\imath)).$$

Hierarchical verification shows that

$$compose_{abs}(B_{IB}(D_1),\ldots,B_{IB}(D_n)) = B_{IB}(Design).$$

Hierarchical verification is valid only if it yields the same result as flat verification (*i.e.*, that paths A → B → C and A → D → C of Figure 10.2 yield the same result). That is, only if

$$\text{ABS}(compose_{con}(B_{con}(D_1),\ldots,B_{con}(D_n))(\imath))$$

$$=$$

$$compose_{abs}(B_{IB}(D_1),\ldots,B_{IB}(D_n))(\text{ABS}(\imath)).$$

We claim that they are equivalent when the satisfaction of *Design*'s constraints imply the satisfaction of *Design*'s components' constraints.

We do not formally prove this claim, to do so would require putting *compose* on much firmer ground. Instead, we sketch a proof of a simplified version of the statement. This sketch uses the function UNABS. UNABS takes an element of the abstract domain and returns an element of the concrete domain. The element chosen doesn't matter as long as $\forall X \in abs[\text{ABS}(\text{UNABS}(X)) = X]$. In the sketch UNABS(ABS(X)) will be substituted for X. In general, this is not a valid substitution. We guarantee, however, that any unabstracted value is later abstracted so that no information is lost.

In the sketch *Design* has N components, D_1 through D_n. We begin with the abstraction of the compositions of the concrete behavior of the components:

$$\text{ABS}(compose_{con}(B_{con}(D_1),\ldots,B_{con}(D_n)(\imath)).$$

The outputs of each component are abstracted then unabstracted:

$$\begin{aligned}\text{ABS}(compose_{con}(&\lambda\, \jmath\,.\,\text{UNABS}(\text{ABS}(B_{con}(D_1)(\jmath))),\\ &\ldots,\\ &\lambda\, \jmath\,.\,\text{UNABS}(\text{ABS}(B_{con}(D_n)(\jmath))))\\ (\imath)).\end{aligned}$$

Each occurrence of $\text{ABS}(B_{abs}(D_i)(\jmath))$ is replaced by D_i's intended abstract behavior. This replacement is only valid when the constraints on D_i are satisfied. For this reason, the satisfaction of the constraints on *Design* together with the physical construction of *Design* must imply the satisfaction of the constraints on each D_i.

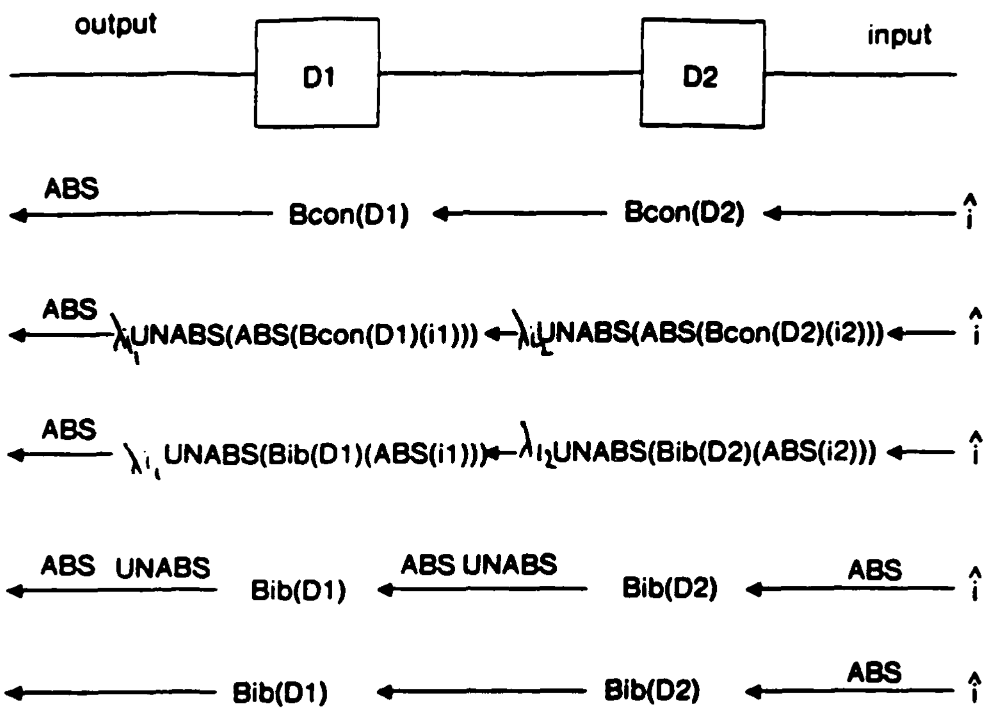

output
D1
D2
input
ABS
Bcon(D1)
Bcon(D2)
î
ABS
λi₁UNABS(ABS(Bcon(D1)(i1)))
λi₂UNABS(ABS(Bcon(D2)(i2)))
î
ABS
λi₁UNABS(Bib(D1)(ABS(i1)))
λi₂UNABS(Bib(D2)(ABS(i2)))
î
ABS UNABS
Bib(D1)
ABS UNABS
Bib(D2)
ABS
î
Bib(D1)
Bib(D2)
ABS
î

Figure 10.3: Deriving $compose_{abs}(B_{IB}(D_1), B_{IB}(D_2))\text{ABS}(\hat{\imath})$ from $\text{ABS}(compose_{con}(B_S(D_1), B_S(D_2))(\hat{\imath}))$

$$\begin{aligned}\text{ABS}(compose_{con}(&\lambda\,\hat{\jmath}\,.\,\text{UNABS}(B_{IB}(D_1)(\text{ABS}(\hat{\jmath}))),\\ &\ldots,\\ &\lambda\,\hat{\jmath}\,.\,\text{UNABS}(B_{IB}(D_n)(\text{ABS}(\hat{\jmath}))))\\ (\hat{\imath})).\end{aligned}$$

All implicit occurrences of ABS(UNABS(x)) are now eliminated. This step has two major results. First, all occurrences of UNABS vanish. Also, the only remaining occurrences of ABS are those on the inputs to the design itself, and not on any internal lines. These occurrences can be collected and moved onto $\hat{\imath}$. Second, because no more ABS's remain on any internal lines, abstract behavior is being composed rather than concrete behavior. These eliminations of UNABS and transferences of ABS yield

$$compose_{abs}(B_{IB}(D_1), \ldots, B_{IB}(D_n))(\text{ABS}(\hat{\imath})),$$

Q.E.D. This proof sketch is shown graphically for a two component design in Figure 10.3.

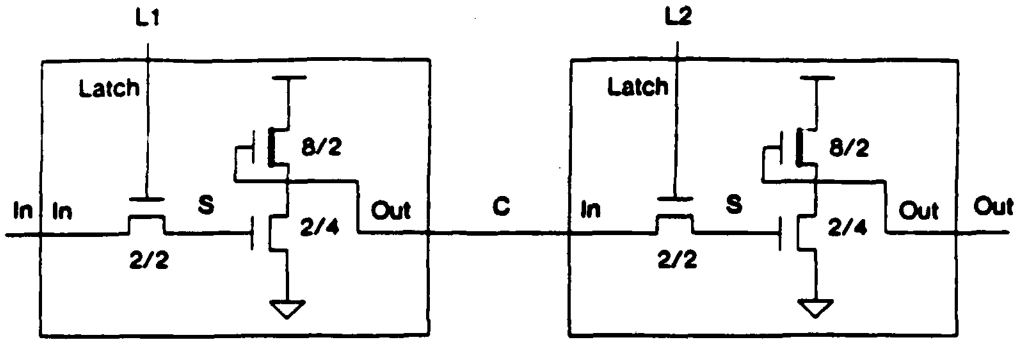


Figure 10.4: A Shift Cell. This circuit is made of two Inverting Latches.

10.4 An Example of Hierarchical Digital Verification

As an example of hierarchical digital verification[2] we will show how a Shift Cell (Figure 1.5) built out of two Inverting Latches is verified. (An overview of the verification of this circuit was shown in Chapter 1.)

The intended digital behavior of the Shift Cell is

```
(df (SHIFT-CELL s1 s2)
  (let ((Left (INVERTING-LATCH s1))
        (Right (INVERTING-LATCH s2)))
    (lambda (in l1 l2)
      (Right (Left in l1) l2))))
```

This digital specification says that the behavior of a Shift Cell is equivalent to two inverting latches, one feeding the other.

There is one logical constraint for the shift cell: tmutex($L1_s$, $L2_s$). It declares that $L1_s$ and $L2_s$ are mutually exclusive. That is, at no time are both $L1_s$ and $L2_s$ asserted. This constraint will be used to satisfy some of the Shift Cell's parts' constraints.

The verification condition is shown "graphically" in Figure 10.5. In this figure P_{found} are the constraints derived during verification and INVERTING LATCH represents the intended digital behavior of the Inverting Latch. The verification condition is shown graphically rather than algebraically to stress that, in Silica Pithecus, the comparison of abstracted behaviors and intended abstract behavior of

[2]Reminder: digital verification is multilevel verification where signal behavior is the concrete behavior and digital behavior is the abstract behavior.

$$\forall\, I_s, L1_s, L2_s, S1_s, S2_s \in S\,(\text{mutex}(L1_s, L2_s) \wedge \bigwedge_{p \in P_{found}} p(I_s, L1_s, L2_s, S1_s, S2_s) \Longrightarrow X = Y)$$

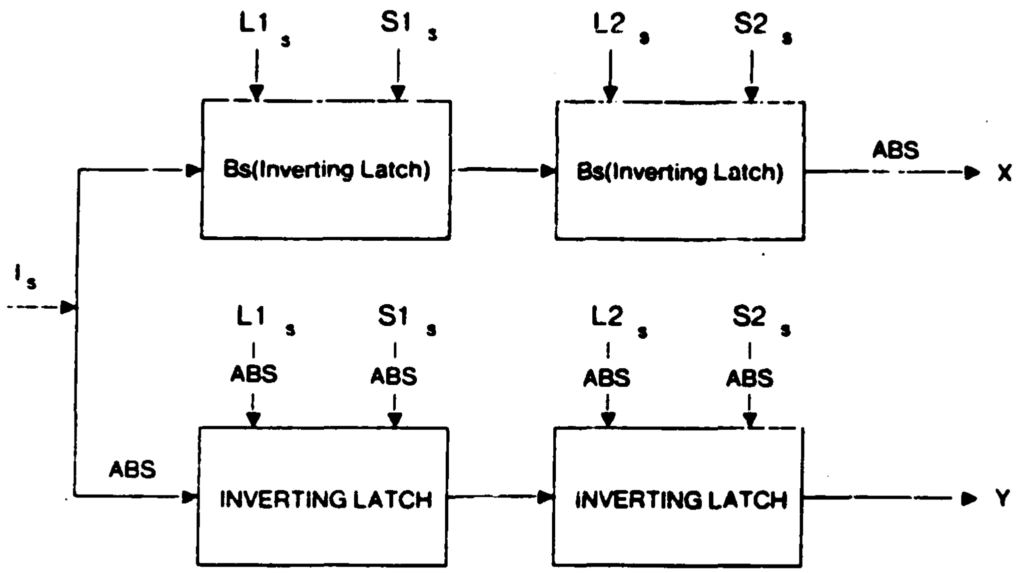


Figure 10.5: The Verification Condition for the Shift Cell. P_{found} are the constraints derived during verification. INVERTING LATCH represents the intended digital behavior of the Inverting Latch.

---

a circuit is done structurally (*cf.* Chapter 6). The verification condition is proved in four steps.

Step 1: Generate the constraints which allow B_S(*Inverting Latch*) to be replaced by B_{IB}(*Inverting Latch*). These are generated by mapping the constraints of each occurrence of the Inverting Latch. Mapping of constraints is similar to the mapping of net behaviors (Section 8.2.1). Table 10.1 shows the generated constraints. (These constraints should be compared with the constraints of the Inverting Latch shown in Figure 1.4.)

Step 2: Process the constraints. Each constraint is either accepted, rejected, or propagated. Chapter 11 discusses processing constraints.

Accepted Constraints: Three of the above nine constraints are satisfied:

- $L2_b \wedge \neg S1_b \Rightarrow$ `overpowers`([Gnd, C]$_C$, [S2]$_{S2}$)
- $L2_b \wedge S1_b \Rightarrow$ `overpowers`([Gnd, Vdd, C]$_C$, [S2]$_{S2}$)
- `falls`($L2_b$) $\Rightarrow$ `falls-first`($L2_s$, C_s).

Mapped Constraints for Left	Mapped Constraints for Right
control($L1_s$)	control($L2_s$)
no-drop($L1_s$)	no-drop($L2_s$)
$L1_b \Rightarrow$ overpowers($[In]_{In}$, $[S1]_{S1}$)	$L2_b \wedge \neg S1_b \Rightarrow$ overpowers($[Gnd, C]_C$, $[S2]_{S2}$)
	$L2_b \wedge S1_b \Rightarrow$ overpowers($[Gnd, Vdd, C]_C$, $[S2]_{S2}$)
falls($L1_b$) $\Rightarrow$ falls-first($L1_s$,In_s)	falls($L2_b$) $\Rightarrow$ falls-first($L2_s$,C_s)

Table 10.1: Mapped Constraints of the Shift Cell

The first two are satisfied because Gnd and Vdd have infinite strength whereas $S2$ has finite strength. The third constraint is satisfied because the logical constraint tmutex($L1_s$, $L2_s$) with the constraints control($L1_s$) and control($L2_s$) ensure that $L2_s$ falls before C_s can change.

Rejected Constraints: None of the constraints are rejected, *i.e.*, shown to not hold.

Propagated Constraints: The remaining six constraints are propagated to become the P_{found} of the proof:

- $L1_b \Rightarrow$ overpowers($[In]_{In}$, $[S1]_{S1}$)
- control($L1_s$)
- control($L2_s$)
- no-drop($L1_s$)
- no-drop($L2_s$)
- falls($L1_b$) $\Rightarrow$ falls-first($L1_s$, In_s)

When the Shift Cell is used these constraints are processed. Constraints are passed up the structural hierarchy until they are satisfied or rejected.

Step 3: Replace each occurrence of B_S(Inverting Latch) by INVERTING LATCH and slide ABS from the output to the inputs.

Step 4: Compare the two structures. They are the same, so the it is concluded that for all boolean inputs $\hat{\imath}$, ABS(B_S(Shift Cell)($\hat{\imath}$)) = B_{IB}(Shift Cell)(ABS($\hat{\imath}$)), *Q.E.D.*

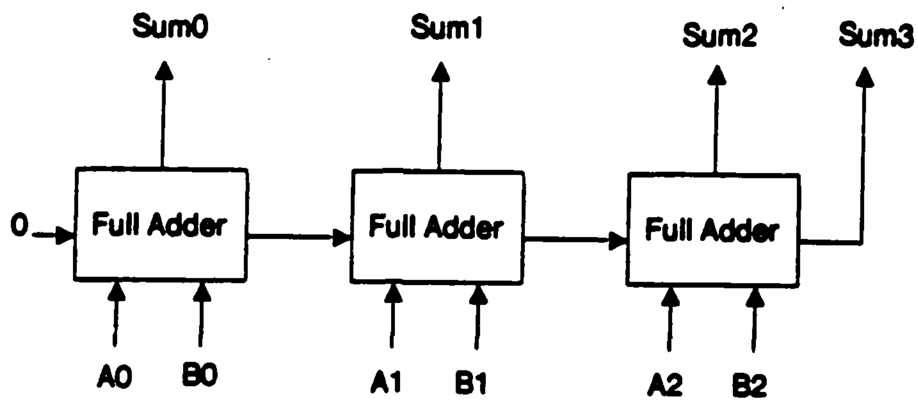


Figure 10.6: 3-Bit Adder

10.5 Problems with Hierarchical Functional Verification

There are many extra problems[3] in performing hierarchical functional verification caused by the richer abstract domains (*e.g.*, sets, numbers, addresses, or a collection of these types) of functional verification. One of the additional difficulties is compound data. Compound data prevents a component's concrete behavior from being replaced by its intended behavior.

10.5.1 Compound Data

We derived the abstract behavior of a design by composing its components' intended abstract behaviors. This was possible because every output of each component had its own abstract behavior. For circuits with ganged outputs (such as vectors of bits) a straightforward substitution of intended abstract behavior for concrete behavior fails.

For example, consider a 3-bit adder which returns a 4-bit result (Figure 10.6). This adder is built out of the adder cell verified in Chapter 3 (Figure 3.3). We proved that the boolean and arithmetic behaviors of the adder cell were related by the equation

$$\begin{gathered}\forall_{i \in B^*}\{Boolean(i_1, i_2, i_3) \Rightarrow \\ \mathrm{ABS}(sum(i_1, i_2, i_3)) + 2\mathrm{ABS}(carry(i_1, i_2, i_3)) \\ = \mathrm{ABS}(i_1) + \mathrm{ABS}(i_2) + \mathrm{ABS}(i_3)\}.\end{gathered}$$

[3]A problem not discussed here is the inadequacy of behaviors as lemmas. Often what is important about a device is the mathematical structure underlying its behavior. When this mathematical structure is not made explicit a verifier will either fail or take a very long time. The interested reader is referred to [Barrow] for a discussion of this problem.

(This equation will be used as a lemma in the verification of the 3-bit adder.) The difficulty is that neither *sum* nor *carry* can be replaced by some function solely of the abstracted inputs to the adder. Either must be replaced in terms of the other.

The intended functional behavior[4] of the 3-bit adder is

$$B_{IB}(\text{3-bit Adder}) = \lambda\ \mathrm{a}\ \mathrm{b}\ .\ a + b.$$

We first generate its concrete behavior from its components' concrete behaviors.[5] The concrete behavior is then abstracted. When comparing the abstracted and intended abstract behaviors the verification condition of the adder cell will be invoked as a lemma as necessary.

The concrete behavior of the 3-bit adder, derived by composing the concrete behavior of its parts, is

$$\begin{aligned}
&B_{B^*}(\text{Adder Cell}) = \\
&\quad \lambda\ [A_0, A_1, A_2, A_3]\ [B_0, B_1, B_2, B_3]\ . \\
&\quad\ [\mathit{Sum}(A_0, B_0, \mathtt{false}), \\
&\quad\ \ \mathit{Sum}(A_1, B_1, \mathit{Carry}(A_0, B_0, \mathtt{false})) \\
&\quad\ \ \mathit{Sum}(A_2, B_2, \mathit{Carry}(A_1, B_1, \mathit{Carry}(A_0, B_0, \mathtt{false}))) \\
&\quad\ \ \mathit{Carry}(A_2, B_2, \mathit{Carry}(A_1, B_1, \mathit{Carry}(A_0, B_0, \mathtt{false})))].
\end{aligned}$$

(This example uses automatic destructuring of formal parameters. This function's inputs are two length four vectors. Destructuring names each bit in the input vectors.)

The abstraction of the adder's concrete behavior is

$$\begin{aligned}
&2^3\mathrm{ABS}(\mathit{Carry}(i_2, j_2, \mathit{Carry}(i_1, j_1, \mathit{Carry}(i_0, j_0, \mathtt{false}))))+ \\
&2^2\mathrm{ABS}(\mathit{Sum}(i_2, j_2, \mathit{Carry}(i_1, j_1, \mathit{Carry}(i_0, j_0, \mathtt{false}))))+ \\
&2^1\mathrm{ABS}(\mathit{Sum}(i_1, j_1, \mathit{Carry}(i_0, j_0, \mathtt{false})))+ \\
&2^0\mathrm{ABS}(\mathit{Sum}(i_0, j_0, \mathtt{false})).
\end{aligned}$$

This must be validated against the sum of the abstraction of the inputs, namely $(2^2\mathrm{ABS}(i_2) + 2\mathrm{ABS}(i_1) + \mathrm{ABS}(i_0)) + (2^2\mathrm{ABS}(j_2) + 2\mathrm{ABS}(j_1) + \mathrm{ABS}(j_0))$.

Therefore, the verification condition to be proved is

$$\forall_{i_1,\ldots,i_n \in B^*}\{\mathit{Boolean}(\mathbf{i}) \Rightarrow$$

$$\begin{aligned}
&2^3\mathrm{ABS}(\mathit{Carry}(i_2, j_2, \mathit{Carry}(i_1, j_1, \mathit{Carry}(i_0, j_0, \mathtt{false}))))+ \\
&2^2\mathrm{ABS}(\mathit{Sum}(i_2, j_2, \mathit{Carry}(i_1, j_1, \mathit{Carry}(i_0, j_0, \mathtt{false}))))+
\end{aligned}$$

[4]This specification is imprecise: what does "+" mean? Fortunately, the adder's output vector is one bit longer than the adder's input vectors so that the normal meaning of + is retained. Had the input and output vector been the same length, then "+" could not have been used. One must be careful to not fall into the trap of using standard operators in non-standard ways.

[5]I find it hard to say whether flat verification or hierarchical verification is being performed. It starts out as flat verification, but, by using lemmas, starts to look like hierarchical verification.

$2^1\mathrm{ABS}(Sum(i_1, j_1, Carry(i_0, j_0, \mathtt{false})))+$
$2^0\mathrm{ABS}(Sum(i_0, j_0, \mathtt{false}))$

$$=$$

$$2^2\mathrm{ABS}(i_2) + 2\mathrm{ABS}(i_1) + \mathrm{ABS}(i_0) + 2^2\mathrm{ABS}(j_2) + 2\mathrm{ABS}(j_1) + \mathrm{ABS}(j_0).$$

There are three alternatives. The first is to expand out the calls to *Sum* and *Carry* and use exhaustive methods to prove the equivalence. Although this strategy would work for the 3-bit adder, it fails as adders get larger. The second alternative is to replace the abstraction of the concrete behaviors of with the intended behavior of the outputs. But as mentioned earlier, this alternatives fails because the outputs are interrelated and are not eliminated by this approach. The third alternative is to be "intelligent" about which of *Sum* and *Carry* to eliminate.

We will rewrite the verification condition of the adder cell as

$$\mathrm{ABS}(sum(i_1, i_2, i_3)) =$$

$$\mathrm{ABS}(i_1) + \mathrm{ABS}(i_2) + \mathrm{ABS}(i_3) - 2\mathrm{ABS}(carry(i_1, i_2, i_3)).$$

This equation is used to eliminate the *Sums* from the verification condition of the 3-bit adder to yield
$8\mathrm{ABS}(Carry(i_2, j_2, Carry(i_1, j_1, Carry(i_0, j_0, \mathtt{false}))))+$
$4\mathrm{ABS}(i_2) + 4\mathrm{ABS}(j_2) + 4\mathrm{ABS}(carry(i_1, j_1, carry(i_0, j_0, \mathtt{false})))-$
$8\mathrm{ABS}(carry(i_2, j_2, carry(i_1, j_1, carry(i_0, j_0, \mathtt{false}))))+$
$2\mathrm{ABS}(i_1) + 2\mathrm{ABS}(j_1) + 2\mathrm{ABS}(carry(i_0, j_0, \mathtt{false}))-$
$4\mathrm{ABS}(carry(i_1, j_1, carry(i_0, j_0, \mathtt{false})))+$
$\mathrm{ABS}(i_0) + \mathrm{ABS}(j_0) + \mathrm{ABS}(\mathtt{false}) - 2\mathrm{ABS}(carry(i_0, j_0, \mathtt{false})).$

$$=$$

$$4\mathrm{ABS}(i_2) + 2\mathrm{ABS}(i_1) + \mathrm{ABS}(i_0) + 4\mathrm{ABS}(j_2) + 2\mathrm{ABS}(j_1) + \mathrm{ABS}(j_0).$$

This is equivalent to

$$\begin{gathered} 4\mathrm{ABS}(i_2) + 4\mathrm{ABS}(j_2) \\ 2\mathrm{ABS}(i_1) + 2\mathrm{ABS}(j_1) \\ \mathrm{ABS}(i_0) + \mathrm{ABS}(j_0) + 0 \\ = \\ 4\mathrm{ABS}(i_2) + 2\mathrm{ABS}(i_1) + \mathrm{ABS}(i_0) + 4\mathrm{ABS}(j_2) + 2\mathrm{ABS}(j_1) + \mathrm{ABS}(j_0), \end{gathered}$$

which is true. The 3-bit adder is therefore correct.

(This example shows that verifying a simple arithmetic unit requires some thought. With the right approach, (*i.e.*, eliminating *Sum*) the proof is simple, without it the proof is considerably more difficult. Because of the thought required, not all verifiers do this proof algebraically. For example, Verify [Barrow] does this proof structurally.[6] This example is indicative of validation at the functional level: often nontrivial theorem proving must be done.)

[6] Verify uses knowledge of the representation of numbers as vectors of booleans to effect the proof: it is hardwired to recognise predetermined structures as correct implementation of adders.

Chapter 11

Constraints

Silica Pithecus's constraint system is the key to Silica Pithecus's hierarchical operation, and therefore the key to Silica Pithecus's speed. Constraints direct a circuit's analysis. They change the nature of analysis from "what does the circuit do?" to "does the circuit do the correct specific thing?" (Such as "do signals flow in right direction here?" or "does signal A fall before signal B?") Such questions are focused and efficient to answer, whereas "what does this circuit do?" is unfocused and inefficient to answer.

Each type of constraint has it own special theorem prover, called a *handler*, for proving instances of the constraint are satisfied. The handlers for logical and threshold constraints are algorithmic and will always get the right answer when they terminate. The handler for flow constraints is heuristic. It runs fast and works for nearly all circuits. The handler for timing constraints (`stable-after` and `falls-first` constraints) is heuristic and fails for circuits which rely on races. The heuristic handlers only generate false negatives when the fail, they never generate false positives.

Constraints are mapped from the level of the component they are on to the level of the design being verified. Constraint mapping mirrors net behavior mapping which was presented in section 8.2.1: nodes and signals are renamed to use names local to the design. Constraint mapping will not be explicitly presented. Examples of constraint mapping appeared in section 10.4.

Each constraint is either *accepted, rejected,* or *propagated*. A constraint is accepted once it is satisfied (shown to hold). A constraint is rejected when it is shown not to hold. Rejected constraints are returned to the designer. A constraint is propagated when it is neither accepted nor rejected. Propagated constraints are attached to the circuit being verified. The propagated constraints will be processed when the circuit is used as a component in a larger circuit.

This chapter discusses each type of constraint in turn.

11.1 Logical Constraints

The two types of logical constraints are steady state logical constraints and temporal logical constraints. Steady state logical constraints predicate boolean relations between the final values of signals.[1] Temporal logical constraints predicate boolean relations between the values of signals at all points in time.

A steady state logical constraint is either a signal name, or a boolean combination (using $\neg$, $\wedge$, and $\vee$) of steady state logical constraints.[2] When just a signal name it used as a steady state logical constraint it asserts that the signal's final value is true. Steady state logical constraints correspond to propositional logic. Some complex predicates are named. For example, mutex(x, y) is shorthand for $(\neg x \wedge \neg y) \vee (\neg x \wedge y) \vee (x \wedge \neg y)$.

The most commonly used temporal logical constraint is tmutex. tmutexasserts that at no time is more than one of its arguments asserted. A special purpose method is used to satisfy temporal logical constraints. We require that all signals in a tmutex constraint be control signals. Then we show the signals are mutex when all the clocks are low and when each clock is high.

Silica Pithecus does not propagate logical constraints, therefore a design must directly satisfy all its components' logical constraints.[3] It can do this either *structurally* or *logically*. A constraint is structurally satisfied when it is satisfied because of the behavior of the components in the design. A constraint is logically satisfied when it is satisfied because of one the design's logical constraints. For example, consider a component C with two inputs x and y and with a steady state logical constraint of $x \vee y$. Figure 11.1 shows this component used in two different designs. Design $D1$ structurally satisfies $x \vee y$ (which maps to $a \vee b$) because $b = \neg a$). Design $D2$ logically satisfies $x \vee y$ (which maps to $c \vee d$) because $c \vee d$ is implied by $D2$'s logical constraint c.

Silica Pithecus does not propagate logical constraints because it wants to maintain user level consistency. For reasons previously discussed, Silica Pithecus does not generate logical constraints, the user must supply them. Logical constraint propagation is similar logical constraint generation: constraints which the designer did not explicitly create get automatically created and attached to designs. It would be confusing to the user to have Silica Pithecus automatically create some constraints but not others. Therefore, Silica Pithecus doesn't propagate logical constraints. (On another note, not propagating logical constraints speeds verification

[1] To be precise, we should say the predicate boolean relations between the *boolean interpretation of* final values of signals. In this chapter we won't be careful about making the distinction between signal values and their boolean interpretations.

[2] There is no way to name new predicates. For example, there is no (defpredicate (at-most-one x y) (not (and x y))).

[3] A design *directly satisfies* a constraint when the constraint doesn't need to be propagated.

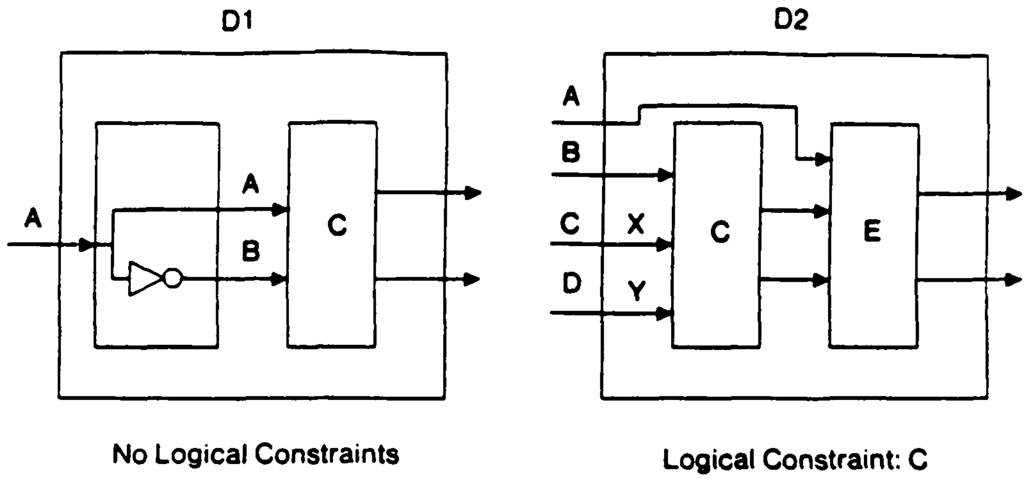


Figure 11.1: Designs must directly satisfy their components' logical constraints. Component C's steady state logical constraint, $x \vee y$, must be satisfied by any design C is in. Design $D1$ structurally satisfies $x \vee y$ (which maps to $a \vee b$) because $b = \neg a$. Design $D2$ logically satisfies $x \vee y$ (which maps to $c \vee d$) because $c \vee d$ is implied by $D2$'s steady state logical constraint c.

and simplifies the implementation.)

Silica Pithecus uses a simple predicate calculus theorem prover to satisfy logical constraints. The kernel of the theorem prover was taken directly from the Lisp 1.5 manual [Lisp 1.5]. The theorem prover, invoked by the procedure **prove**, has two inputs: the predicate to be proved, and an environment of assertions. **prove** returns **true** if the predicate can be proved from the assertions, otherwise it returns `false`. The following algorithm processes steady state logical constraints.

Algorithm Process Steady State Logical Constraints:

Inputs: Design D, its components' mapped steady state logical constraints CLC, and its own steady state logical constraints LC.

Outputs: A list of the unsatisfied CLC.

Method: Label each wire (node) W_i between components of D with the boolean formula representing the value on W_i. For each CLC_j, replace each wire (node) name with the wire's label to yield a new constraint CLC'_j. If **prove**(CLC'_j, LC) returns false add CLC_j to the list of unsatisfied steady state logical constraints.

For example, consider again the designs of Figure 11.1. These designs have their wires labelled according to the boolean functions output by components driving the wire. C's mapped steady state logical constraint, $a \vee b$, is satisfied by $D1$ because **prove**$(a \vee \neg a, \{\})$ returns **true**. Similarly, C's mapped steady state logical constraint, $c \vee d$, is satisfied by $D2$ because **prove**$(c \vee d, \{c\})$ returns **true**.

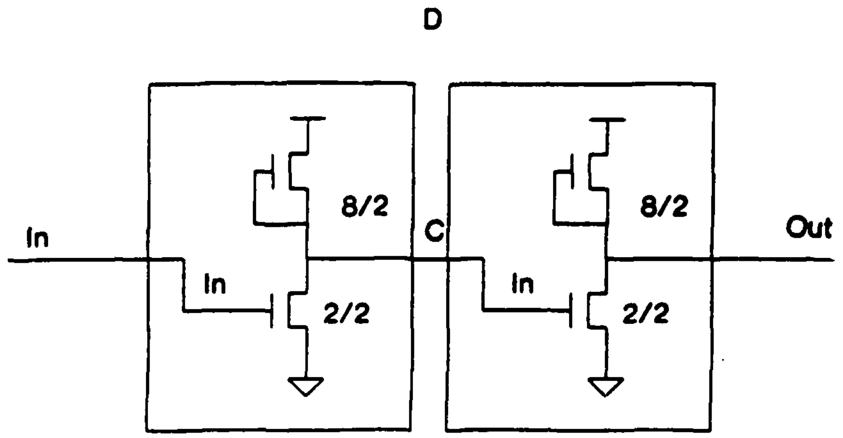


Figure 11.2: Processing Threshold Constraints

11.2 Threshold Constraints

A threshold constraint on a signal is satisfied by showing that every final net computing the signal yields either a voltage of 5 volts or a voltage 1.5 volts. For example, consider the output of an inverter *A* driving the input of a minimally sized inverter *B* (Figure 11.2). Because *B* is minimally sized, there is a threshold constraint on its input. *B*'s mapped constraint is no-drop(Internal). The net behavior of Internal is the mapped net behavior of *A*'s Out node. This net behavior is:

$$\begin{aligned} \mathrm{Out}_{net} = \lambda\ t\ .\ & \mathrm{In}_b(t) \rightarrow [\mathrm{Vdd\ Out\ Gnd}], \\ & \mathtt{NOT}(\mathrm{In}_b(t)) \rightarrow [\mathrm{Vdd\ Out}]. \end{aligned}$$

The first net yields 1 volt, the second yields 5 volts. Therefore the threshold constraint is satisfied.

(A note on the implementation: Silica Pithecus doesn't check threshold constraints by examining net behaviors during constraint processing. Instead, checking whether a node puts out an undegraded signal is done at abstraction time and the result remembered. This saves repeated checking that would occur for a gate with large fanout.)

11.3 Flow Constraints

Mapping flow constraints is complex and expensive, but processing the mapped constraints is simple. Each is discussed in turn.

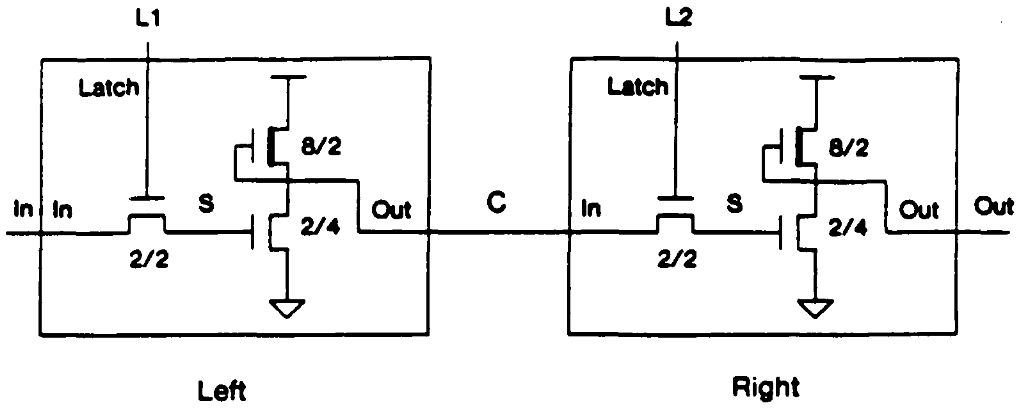


Figure 11.3: A Shift Cell

We will use the Shift Cell of Chapters 1 and 10 (Figure 11.3) as an example of mapping and processing flow constraints. This device has two components, Left and Right, which are Inverting Latches. It also has a temporal logical constraint of $\mathtt{tmutex}(L1, L2)$. The Inverting Latch has a flow constraint of

$$\text{Latch}_{\Downarrow} \Rightarrow \mathtt{overpowers}([\text{In}]_{In}, [\text{S}]_S).$$

(A note on the implementation: constraint mapping and processing are interleaved. This prevents extraneous work and extraneous net expansion. Interested readers are invited to look at the code.)

11.3.1 Mapping Flow Constraints

Flow constraints are mapped in two steps:

1. Rename the nodes used in the constraints.

2. Expand nets named in the constraints.

The first part, renaming, corresponds to what we have been calling mapping for the other constraints: names local to the component are replaced with names used by the design. The second part, expansion, is complex: it finds all nets which go into and out of an input node. When a constraint is expanded, it may turn into multiple constraints. (For example, consider the mapped flow constraints in Table 10.1.)

Renaming Flow Constraints

A flow constraint is renamed exactly as a net behavior expression is renamed (*cf.* Section 8.2.1). We will show how the flow constraints of the Shift Cell's two components is renamed. The Inverting Latch's flow constraint is

$$\text{Latch}_b \Rightarrow \textbf{overpowers}([\text{In}]_{In}, [\text{S}]_S).$$

This is renamed to

$$\text{L1}_b \Rightarrow \textbf{overpowers}([\text{In}]_{In}, [(\text{S Left})]_{(SLeft)}))$$

for the Left Inverting Latch, and renamed to

$$\text{L1}_b \Rightarrow \textbf{overpowers}([\text{C}]_C, [(\text{S Right})]_{(SRight)})$$

for the Right Inverting Latch. The notation (<node-name> <instance-name>) is used to name components' internal nodes (*cf.* Section 6.2).

Expanding Flow Constraints

The basic idea is to find the nets computing signals going into an input node, and the nets that those signals must drive. Once each set of nets are found, their cross-product is found and becomes the mapped constraints.

For example, consider the Left's output signal. There are two nets which compute this signal, [Vdd, Out, C], and [Vdd, Gnd, Out, C]. These nets occur on $\neg(\text{S Left})_b$ and $(\text{S Left})_b$, respectively. There is only one net to be overpowered, [(S Right)]. The cross product of the "drivers" with the "drivees" yields the two constraints

- $\text{L2}_b \wedge \neg(\text{S Left})_b \Rightarrow \textbf{overpowers}([\text{Vdd, C}]_C, [(\text{S Right})]_{(SRight)})$
- $\text{L2}_b \wedge (\text{S Left})_b \Rightarrow \textbf{overpowers}([\text{Gnd, Vdd, C}]_C, [(\text{S Right})]_{(SRight)})$.

We now discuss the details of expanding flow constraints (*i.e.*, how to locate the "driver" and and "drivee" nets). Expanding flow constraints is similar to the second and third steps (net merging and net expansion) of net behavior generation (Chapter 8). Constraint expansion differs in that after merging, certain nodes are dropped from nets before expansion to avoid having the expansion include the nets the must be overpowered (or must do the overpowering).

Consider the constraint $P \Rightarrow \textbf{overpowers}(Net1_{Node1}, Net2_{Node2})$. The following rules find the "drivers."

1. Perform Net Renaming and Merging for node *Node*1 to yield X.
2. Delete all nodes from *Net*2 from all nets of X.
3. Perform Net Expansion on X.

Analogous rules are used to find the "drivees." The cross product of the two net behavior expression is then generated. For every possible combination of a clause $P1 \rightarrow Net1$ from the "drivers" and a clause $P2 \rightarrow Net2$ from the "drivees," the constraint $P \wedge P1 \wedge P2 \rightarrow \mathbf{overpowers}(Net1, Net2)$ is generated. If $P \wedge P1 \wedge P2$ is logically disallowed then the constraint is ignored.

Using these rules, the following flows constraints of the Shift Cell are the result of mapping the flow constraints of the Shift Cell's components:

$$\mathrm{L1}_b \Rightarrow \mathbf{overpowers}([\mathrm{In}]_{In}, [(\mathrm{S\ Left})]_{(SLeft)})$$

$$\mathrm{L2}_b \wedge \neg(\mathrm{S\ Left})_b \Rightarrow \mathbf{overpowers}([\mathrm{Vdd,\ C}]_C, [(\mathrm{S\ Right})]_{(SRight)})$$

$$\mathrm{L2}_b \wedge (\mathrm{S\ Left})_b \Rightarrow \mathbf{overpowers}([\mathrm{Gnd,\ Vdd,\ C}]_C, [(\mathrm{S\ Right})]_{(SRight)})$$

11.3.2 Processing Flow Constraints

The result of constraint mapping is a set of flow constraints. Each has the form $\mathbf{overpowers}(Net1, Net2)$. Not considering input/output nodes, an **overpowers** constraint is accepted when the combined strength of the nodes $Net1$ is ten times greater than the combined strength of the nodes in $Net2$. Otherwise the constraint is rejected and a charge sharing error returned to the designer.

When either net of a flow constraint contains an I/O node (*i.e.*, a node that is either an input or an output of the circuit being verified) then the constraint may be propagated rather than accepted or rejected. Specifically, a flow constraint cannot be accepted if the net to be overpowered contains an I/O node, and it cannot be rejected if the overpowering net contains an I/O node.

11.3.3 Another Example: A Faulty Circuit

As an example of a constraint that cannot be satisfied because of a buggy circuit consider connecting the Latched Inverter to the Inverting Latch, shown in Figure 11.4. The constraint to be mapped is the same as in the previous example:

$$\mathrm{Latch}_b \Rightarrow \mathbf{overpowers}([\mathrm{In}]_{In}, [\mathrm{S}]_S).$$

The net behavior of the C node is

$$\begin{aligned} C_{net} = \lambda\ t\ .\ & \mathrm{L1}_b(t) \wedge \mathrm{In}_b(t) \rightarrow [\mathrm{Gnd\ Vdd\ C\ (S\ left)}], \\ & \mathrm{L1}_b(t) \wedge \neg\ \mathrm{In}_b(t) \rightarrow [\mathrm{Vdd\ C\ (S\ left)}], \\ & \neg\ \mathrm{L1}_b(t) \rightarrow [\mathrm{C}]. \end{aligned}$$

Therefore the constraint maps to the following three constraints:

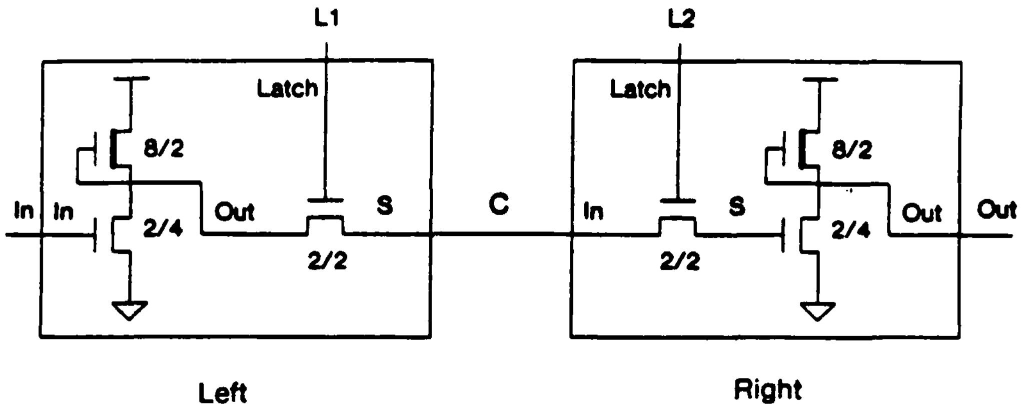


Figure 11.4: A Faulty Circuit. The error in this circuit is caught by a rejected flow constraint.

- $\mathrm{L2}_b \wedge \mathrm{L1}_b \wedge \mathrm{In}_b \Rightarrow$ **overpowers**$([\mathrm{Gnd\ Vdd\ C\ (S\ left)}]_C, [\mathrm{(S\ Right)}]_{(SRight)})$.
- $\mathrm{L2}_b \wedge \mathrm{L1}_b \wedge \neg\mathrm{In}_b \Rightarrow$ **overpowers**$([\mathrm{Vdd\ C\ (S\ left)}]_C, [\mathrm{(S\ Right)}]_{(SRight)})$.
- $\mathrm{L2}_b \wedge \neg\mathrm{L1}_b \Rightarrow$ **overpowers**$([\mathrm{C}]_C, [\mathrm{(S\ Right)}]_{(SRight)})$.

When the last clause is true (*i.e.*, $\mathrm{L1}_b$ is **false** and $\mathrm{L2}_b$ is **true**), then $[\mathrm{C}]_C$ must overpower $[\mathrm{(S\ Right)}]_{(SRight)})$. Unfortunately, both C and (S Right) have the same capacitance, so this constraint is not satisfied. Silica Pithecus reports this as a charge sharing bug.

Note that had there been a logical constraint of, for example, either $(L1 \wedge L2) \vee (\neg L1 \wedge \neg L2)$ or $\neg(\neg L1 \wedge L2)$, then the erring condition would not have arisen, and no error message would have been generated.

Chapter 12

Future Research

This research engenders two kinds of future research: extending the theory and implementation, and looking at old problems in new ways.

12.1 Combinatorial Blowup

The major (pragmatic) deficiency is the brute force nature of the theory. There are many common circuits for which the number of possible nets is large. For example, the output node of a 32-input nor gate has 2^{32} possible different nets. The outputs of large PLA's have even more possible nets. To overcome this problem special code must be written for gates with large numbers of inputs. This code would only generate the minimal number of nets needed to catch ratio and threshold problems.

Combinatorial blowup also strikes when discharging logical constraints. When the inputs to some device comes from a large PLA, discharging the logical constraints can be very expensive. A smarter theorem prover must be employed which has heuristics for quickly proving constraints hold.

I believe that there are only a few general areas where combinatorial problems bite and that special case code can be written to solve 95% of the problems.

12.2 Feedback

Feedback was completely ignored. Silica Pithecus can detect static feedback. However, it does not detect transient feedback. I do not believe that making it detect transient feedback is hard, it just hasn't been done.

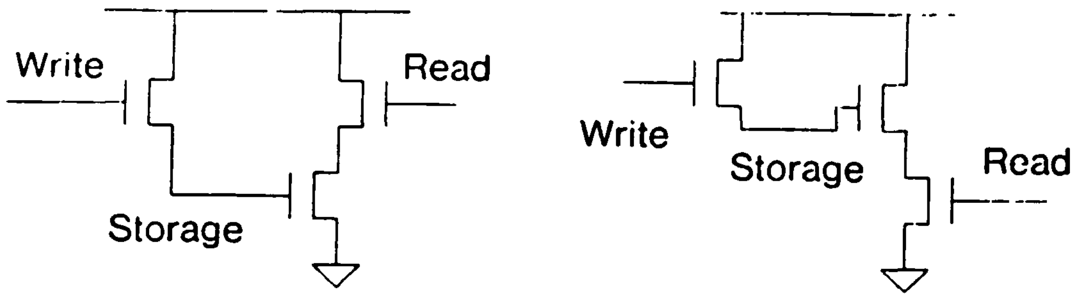


Figure 12.1: Two Three-Transistor RAMS

12.3 Circuit Timing

A timing estimator/verifier built on top of Silica Pithecus would be more accurate and easier to use than other static verifiers such as Crystal [Crystal] or TV [TV]. Because the logical structure and signal flow of a circuit is accurately known, there is more information to exploit to avoid approximations and potentially failing heuristics.

For example, a timing system embedded in Silica Pithecus would not need special scans or treewalks of the circuit to determining the "direction" of transistors. Other timing systems either scan the circuit or use hints from the designer to determine the direction of transistors. Crystal uses hints from the designer. Unfortunately, these hints are not checked. The hints to Crystal are similar in intent to Silica Pithecus's logical constraints. Silica Pithecus checks the designer's logical constraints, however, and Crystal doesn't check the designer's hints. If the designer gives Crystal the wrong hints he will get back the wrong answers. Also, TV's determination of transistor directions is not guaranteed to be correct. If the determination is incorrect wrong timing results will be obtained.

What confuses other timing verifiers is not knowing the possible nets of a circuit. Because Silica Pithecus keeps track of possible nets with a vengeance, timing analysis and critical path analysis can be very precise.

12.4 Capacitance Coupling

Ignoring capacitive coupling causes an important class of bugs to be ignored and prevents an important class of circuits from being verified.

For example, consider the two three-transistor RAMS in Figure 12.1. The cell on the left will work, but the one on the right will not work. It fails due to capacitive

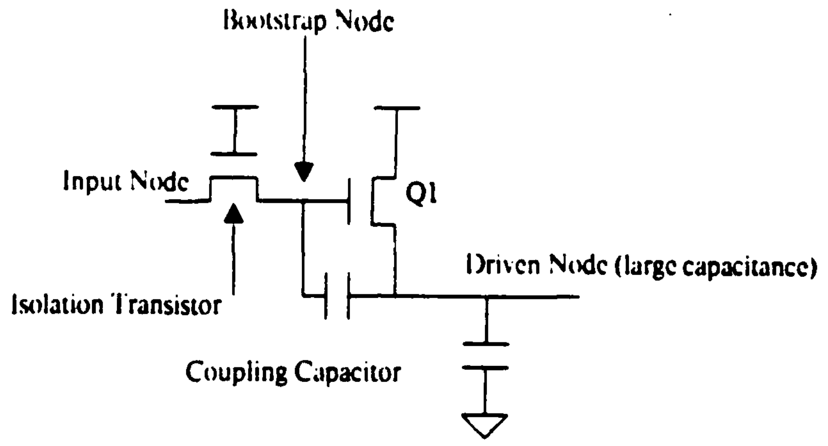


Figure 12.2: Bootstrap Driver. This device has much faster response and drive than expected because of capacitive coupling driving critical nodes high.

coupling: it is impossible to store a high value on the storage node. Silica Pithecus does not detect this bug.

As an example of a circuit we would like to be able to verify, consider a *bootstrap driver* (Figure 12.2), which is used in time-critical applications. Assume that the bootstrap node, driven node, and input node are initially at ground. The driven node's capacitance is much greater than the bootstrap node's capacitance. The bootstrap driver operates as follows:

1. When the input node rises to Vdd the bootstrap node rises with it. The driven node does not begin to rise until after the bootstrap node exceeds the threshold for Q1. The driven node does not rise as quickly as the bootstrap node because of its greater capacitance.

2. At some point after Q1 turns on, the isolation transistor turns off, isolating the bootstrap node. The isolation transistor turns off when the bootstrap node rises to within a threshold voltage of the input node. The isolation transistor will remain off even if the bootstrap node continues to rise. At this time there is a voltage differential across the coupling capacitor.

3. After this point, as the driven node continues to rise the capacitor maintains the voltage differential produced above thereby driving the bootstrap node even higher.

4. Because the isolation transistor remains off, the bootstrap node can be driven

fairly high. In some configurations it can reach eight volts.[1]

5. As the bootstrap node goes up the voltage on the gate of the pullup goes up well beyond five volts thereby allowing the driven node to reach Vdd.

As a result of this process the driven node rises much faster and higher than if the input node were connected directly to the driven node. A system ignoring capacitive coupling calculates an incorrect final voltage (two threshold drops below Vdd) for the driven node, as well as an incorrect estimate of the time it takes to stabilize.

The bootstrap driver is not a digital circuit according the definition given in Chapter 1. According to that definition, a circuit is not digital when the rate of change of voltage affects the final state of a circuit. The correct operation of the bootstrap driver requires the bootstrap node to rise much faster than the driven node. If the driven node rises faster, then there is no "bootstrap" action.

I don't know how to extend the theory to include bootstrap drivers and similar circuits. This research should be done.

12.5 Multiple Abstraction Functions

We concentrated on avoiding stored charge and ensuring voltage levels were adequate. An abstraction was formulated which ensured these two properties. It might be the case that more analog-like circuits, such as bootstrap devices and sense amplifiers, could be verified by positing the proper abstraction functions. Ideally, representations of circuit behavior could be developed which make invalid signals apparent.

When a system contains multiple abstraction functions the user would have to declare which abstraction functions go with which nodes. There might be heuristics which allow a system to guess correctly 95% of the time and to otherwise ask the user.

[1] "Hot Clock nMOS" which uses 7 volt clocks and 5 volt Vdd has achieved bootstrap voltages of 11 volts [Seitz].

References and Bibliography

[Abelson] Harold Abelson and Gerald J. Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs* MIT Press 1985.

[Baker] Clark M. Baker and Chris Terman, "Tools for verifying integrated circuit designs," *Lambda*, Fourth Quarter, 1980.

[Barrow] Harry Barrow. "Proving the correctness of digital hardware designs" VLSI Design, July 1984

[Bobrow] Daniel Bobrow, Editor, *Qualitative Reasoning about Physical Systems,* MIT Press, 1985.

[Boyer] Robert S. Boyer and J. Strother Moore, *A Computational Logic,* Academic Press, 1979.

[Bryant81] Randal Bryant, *A Switch-Level Simulation Model for Integrated Logic Circuits,* MIT Laboratory for Computer Science Technical Report TR-259. PhD. 1981

[Bryant83] Randal Bryant, "Race detection in MOS circuits by ternary simulation," VLSI 83, F. Anceau and E.J. Aaas (editors), pages 85-95, 1983

[Bryant85] Randal Bryant, "Symbolic Verification of MOS Circuits" Proceedings from the 1985 Chapel Hill Conference on VLSI. Edited by Henry Fuchs. Computer Science Press. 1985

[Bryant86] Randal Bryant, "Can a Simulator Verify a Circuit?" In *Formal Aspects of VLSI Design,* G.J. Milne, Editor, North-Holland, 1986

[De Kleer] Johan de Kleer and John Seely Brown, "A Qualitative Physics Based on Confluences," In *Qualitative Reasoning about Physical Systems,* Daniel Bobrow, Editor MIT Press, 1985.

[DPL] Batali, J. and Hartheimer A. *The Design Procedure Language Manual,* MIT Artificial Intelligence Laboratory Memo 598, 1980

[Eveking] Hans Eveking, "The verification of mutilevel hardware descriptions," Unpublished proceedings of the Darmstadt Workshop on the Verification of Hardware Designs.

[Fujita] Masihiro Fujita, Hidehiko Tanaka, Tohru Moto-oka, "Verification with Prolog and temporal logic," *Computer Hardware Description Languages and their Application*, Uehara and Barbacci (Editors), North-Holland Publishing Company, 1983

[German] Steven German, "Zeus, a language for expressing algorithms in hardware," *IEEE Computer* 18(2), February, 1985

[Gordon81] Mike Gordon. "A Very Simple Model of Sequential Behavior of nMOS," Proceedings VLSI International Conference, J Gray (ed.) Academic Press, London and New York 1981

[Gordon83] "Proving a Computer Correct" University of Cambridge computer laboratory Technical Report No. 42. 1983

[Gordon84a] Personal Communication.

[Gordon84b] "How to Specify and Verify Hardware Using Higher Order Logic" Unpublished Lecture Notes, Autumn 1984. Cambridge University, UK. 1984

[Gordon84c] "Multilevel verification using higher order logic," Unpublished proceedings of the Darmstadt Workshop on the Verification of Hardware Designs.

[Hanes] L. H. Hanes, *Logic Design Verification Using Static Analysis*, University of Illinois, PhD. 1983

[Hunt] Warren Hunt, *FM8501: A Verified Microprocessor*, University of Texas at Austin, PhD., Institute for Computing Science Technical Report 47, 1985.

[Johnson] Steven Johnson, *Synthesis of Digital Designs from Recursion Equations*, Mit Press, 1984.

[Jouppi] Norman Jouppi, *Timing Verification and Performance Improvement of MOS VLSI Designs* Stanford University Computer Systems Laboratory Technical Report No. 84-266 PhD. 1984

[Karplus] Kevin Karplus, "Exclusion constraints, a new application of graph algorithms to VLSI Design," Proceedings of the Fourth MIT Conference on Advanced Research in VLSI, Pages 123 - 139, 1986

[Krambeck] R. H. Krambeck, Charles M. Lee, and Hung-Fai S. Law, "High-speed compact circuits with CMOS" IEEE Journal of Solid-State Circuits. SC-17(3). Pages 614-618. June 1982

[Lin] Tzu-Mu Lin, and C. Mead, "Signal delay in general RC networks with application to timing simulation of digital integrated circuits" 1984 Conference on Advanced Research in VLSI, MIT.

[Lisp 1.5] The Lisp 1.5 Manual, John McCarthy, MIT Press, 1963

[McWilliams] Thomas McWilliams, *Verification of Timing Constraints on Large Digital Systems.* Lawrence Livermore National Laboratory Report UCRL-52995 (PhD Thesis) 1980.

[Mead/Conway] *Introduction to VLSI Systems*, Addison Wesley, 1980

[Mishra/Clarke] B. Mishra, and E.M. Clarke, "Automatic and hierarchical verification of asynchronous circuits using temporal logic." CMU Tech Report CMU-CS-83-115 1983

[Moszkowski] Ben Moszkowski, *Reasoning about digital circuits*, Stanford Univerity Department of Computer Science Tech Report STAN-CS-83-970. 1983 (Phd Thesis)

[Näher] T. Lengauer and S. Näher, *Delay-independent switch-level simulation of digital mos circuits*, Technical Report TR 03/84, SFB 124-B2, University d. Saarlandes, 1984.

[Ousterhout] J. Ousterhout, "Crystal: A timing analyzer for nMOS circuits," Proceedings from the Third Caltech Conference on VLSI. Edited by Randal Bryant. Computer Science Press. 1983

[Patel] Dorab Patel, Martine Schlag, and Milos Ercegovac, "vFP: An environment for the multi-level specification, analysis, and synthesis of hardward algorithms," *Functional Programming Languages and Computer Architecture*, Springer-Verlag Lecture Notes in Computer Science 201, 1985

[Penfield] Paul Penfield, J. Rubenstein, and Mark Horowitz, "Signal delays in RC tree networks," IEEE Trans. on Computer Aided Design, 1983 pp. 269-283 1981

[Seitz] Personal Communication.

[Seitz *et. al.*] C. Seitz, A. Frey, S. Mattisson, S. Rabin, D. Speck, and J. van de Snepscheut, "Hot-Clock nMOS," Proc of the 1985 Chapel Hill Conference on VLSI. Henry Fuchs, Editor. Computer Science Press 1985

[Sheeran] Mary Sheeran, "muFP, a language for VLSI design," *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pp. 104–112, 1984

[Siskind] Siskind, Jeff. Personal Communication.

[Steele] Guy L. Steele, *RABBIT: A Compiler for Scheme*, MIT Artificial Intelligence Laboratory Technical Report AI-TR-474. (Masters Thesis) 1979

[Terman] Christopher J. Terman, *Simulation Tools for Digital Design*, MIT Laboratory for Computer Science Report TR-304 (PhD Thesis) 1983

[TV] Norman P. Jouppi, "TV: An nMOS timing analyzer," Proceedings of the 3rd Caltech VLSI Conference, pp 71-86, March 1983

[Verify] See [Barrow].

[Wagner] T. J. Wagner, *Hardware Verification*, Stanford Computer Science Department STAN-CS-77-632. (PhD) 1977

[Weise] Daniel W. Weise, *Exploiting Hierarchy in the Analysis of VLSI Systems*, MIT VLSI Memo 82-104 (Masters Thesis) 1982

[Weste] Niel Weste, and Kamran Eshraghian, *Principles of CMOS VLSI Design, A Systems Perspective*, Addison Wesley, 1985.

[Whitney] Telle Whitney, *A Hierarchical Design-Rule Checker,* Masters Thesis, California Institute of Technology 1981.

[Williams] Brian Williams, "Qualitative analysis of MOS circuits," Artificial Intelligence (24). Pages 281-346 1984

[Wojcik] Anthony S. Wojcik, "Formal design verification of digital systems," Proceedings of the 20th Design Automation Conference 1983.